GraphDB Documentation

Release 11.3.3

Ontotext

28 September 2026

CONTENTS

1 Getting started with GraphDB **1**
1.1 What is GraphDB? . . . 1
1.1.1 Licensing . . . 2
1.1.1.1 GraphDB feature comparison . . . 3
1.1.2 Cloud deployments for GraphDB . . . 4
1.2 How to install GraphDB . . . 4
1.2.1 Requirements . . . 5
1.2.1.1 Minimum requirements . . . 5
1.2.1.2 Hardware sizing . . . 5
1.2.1.3 Memory management . . . 6
1.2.2 GraphDB Desktop installation . . . 7
1.2.2.1 On Windows . . . 7
1.2.2.2 On MacOS . . . 7
1.2.2.3 On Linux . . . 7
1.2.2.4 Configuring your running GraphDB instance . . . 8
1.2.2.5 Stopping GraphDB . . . 9
1.2.3 GraphDB standalone server . . . 9
1.2.3.1 Running GraphDB . . . 9
1.2.3.2 Configuring GraphDB . . . 10
1.2.3.3 Stopping the database server . . . 11
1.2.4 Structure of the distribution package . . . 11
1.3 Working with Workbench . . . 12
1.3.1 Create your first repository using Workbench . . . 15
1.3.2 Load and query some data . . . 16
1.3.2.1 Query data through the Workbench . . . 16
1.3.2.2 Query data programmatically . . . 17
1.3.3 Customizing Workbench behavior . . . 18
1.3.3.1 User settings . . . 19
1.3.3.2 Workbench settings . . . 19
1.3.4 Autocomplete index . . . 20
1.3.4.1 How the index works . . . 21
1.3.4.2 Autocomplete in the SPARQL editor . . . 22
1.3.4.3 Autocomplete in the View resource box . . . 23
1.3.4.4 Using SPARQL queries to work with the autocomplete index . . . 24
1.3.5 The Workbench notification area . . . 24

2 Managing repositories **27**
2.1 Overview of repositories . . . 27
2.2 Creating and editing repositories . . . 27
2.2.1 Creating a repository . . . 28
2.2.1.1 Using Workbench . . . 28
2.2.2 Using a repository . . . 29
2.2.2.1 Connecting to a repository . . . 29
2.2.2.2 Making it the default repository . . . 29

2.2.2.3 Restarting a repository . . . 30
2.3 Configuring a repository . . . 30
2.3.1 Plan a repository configuration . . . 30
2.3.2 Configure a repository through the GraphDB Workbench . . . 30
2.3.3 Edit a repository configuration using Workbench . . . 32
2.3.4 Configure a repository programmatically . . . 32
2.3.5 Configuration parameters . . . 33
2.3.6 Namespaces defined for the repository . . . 35
2.3.7 Reconfigure a repository . . . 37
2.3.7.1 Using Workbench . . . 37
2.3.7.2 Global overrides . . . 37
2.3.8 Rename a repository . . . 37
2.3.8.1 Using Workbench . . . 37
2.3.9 Change location settings . . . 37
2.3.10 Repository Caching Mechanism . . . 38
2.3.10.1 Limiting the number of active repositories . . . 38
2.3.10.2 Set a timeout for active repositories . . . 39
2.4 Connecting to remote locations and data sources . . . 39
2.4.1 Creating a repository on an attached remote GraphDB location . . . 40
2.5 Using plugins . . . 40
2.5.1 Activate or deactivate plugins . . . 41
2.5.1.1 From the Workbench . . . 41
2.5.1.2 With a SPARQL query . . . 42
2.5.1.3 List plugins and their state . . . 43
2.5.2 Enable or disable plugins . . . 43
2.5.2.1 Autocomplete index . . . 43
2.5.2.2 Change tracking . . . 43
2.5.2.3 Data history and versioning . . . 43
2.5.2.4 GeoSPARQL support . . . 43
2.5.2.5 RDF Rank . . . 43
2.6 Inferencing . . . 43
2.6.1 Inference in GraphDB . . . 43
2.6.1.1 Standard rulesets . . . 44
2.6.1.2 Custom rulesets . . . 44
2.6.2 Checking inference derivation with the Proof plugin . . . 44
2.6.2.1 Predicates and namespace . . . 45
2.6.2.2 Usage and examples . . . 45
2.6.3 Query time inference with the Provenance plugin . . . 54
2.6.3.1 Predicates . . . 55
2.6.3.2 Registering the plugin . . . 55
2.6.3.3 Usage and examples . . . 55
2.7 Data storage . . . 57
2.7.1 GraphDB persistence strategy . . . 57
2.7.2 GraphDB indexing options . . . 57
2.7.2.1 Transaction control . . . 57
2.7.2.2 Predicate lists . . . 58
2.7.2.3 Context index . . . 58
2.7.2.4 Literal index . . . 58
2.7.2.5 Handling of explicit and implicit statements . . . 59
2.8 Monitoring and terminating queries and update requests . . . 60
2.8.1 Query monitoring and termination using the Workbench . . . 60
2.8.2 Automatically prevent long running queries . . . 61
2.9 Accessing relational databases with data virtualization . . . 61
2.9.1 Overview and features . . . 61
2.9.2 Setup and configuration . . . 62
2.9.2.1 JDBC driver . . . 62
2.9.2.2 Configuration files . . . 62
2.9.3 Creating a virtual repository from the Workbench . . . 63

2.9.3.1 With a generic JDBC driver . . . 64
2.9.3.2 With one of the other supported database drivers . . . 65
2.9.3.3 Creating a virtual repository using curl . . . 65
2.9.4 Mapping language . . . 66
2.9.5 SPARQL endpoint . . . 66
2.9.6 Query federation . . . 66
2.9.7 Usage scenario . . . 66
2.9.7.1 SPARQL example queries . . . 68
2.9.7.2 Query federation example . . . 68
2.9.8 Limitations . . . 70
2.10 Federation from multiple SPARQL endpoints with FedX . . . 70
2.10.1 Overview . . . 70
2.10.2 Features . . . 70
2.10.3 Usage scenarios . . . 71
2.10.3.1 Linked data cloud exploration . . . 71
2.10.3.2 Virtual knowledge graph over local native and RDBMS-based repositories . . . 76
2.10.3.3 FedX with enabled GraphDB security . . . 79
2.10.4 Configuration parameters . . . 84
2.10.5 Limitations . . . 85
2.11 Graph replacement optimization . . . 85
2.12 Managing repositories and locations via the GraphDB REST API . . . 86
2.12.1 Managing repositories . . . 87
2.12.1.1 Create a repository . . . 87
2.12.1.2 List repositories . . . 87
2.12.1.3 Retrieve information about one repository . . . 88
2.12.1.4 Count the triples in a repository . . . 88
2.12.1.5 Restart a repository . . . 88
2.12.1.6 Delete a repository . . . 88
2.12.1.7 Edit a repository's configuration . . . 88
2.12.2 Managing locations . . . 92
2.12.2.1 Attach a location . . . 92
2.12.2.2 List locations . . . 93
2.12.2.3 Detach a location . . . 93
2.12.3 Further reading . . . 93

3 Loading and exploring data **95**
3.1 Loading data . . . 95
3.1.1 Loading data using Workbench . . . 95
3.1.1.1 Importing local RDF files . . . 96
3.1.1.2 Importing remote RDF files . . . 97
3.1.1.3 Importing RDF data from a text snippet . . . 97
3.1.1.4 Importing server files . . . 97
3.1.1.5 Import settings . . . 97
3.1.1.6 Import data with an INSERT query using the SPARQL endpoint . . . 98
3.1.2 Loading data using the ImportRDF tool . . . 99
3.1.2.1 Load vs Preload . . . 99
3.1.2.2 Command line options . . . 99
3.1.2.3 Loading data . . . 99
3.1.2.4 Repository configuration template . . . 101
3.1.2.5 Tuning load . . . 101
3.1.2.6 Tuning preload . . . 102
3.1.2.7 Resuming data loading with preload . . . 102
3.1.3 Updating data with SPARQL templates . . . 102
3.1.3.1 Overview . . . 102
3.1.3.2 Transport mechanisms . . . 103
3.1.3.3 SPARQL templates . . . 105
3.1.4 SHACL validation . . . 111
3.1.4.1 What is SHACL validation? . . . 111

3.1.4.2 Usage . . . 111
3.1.4.3 Validation logging and report . . . 120
3.1.4.4 Supported SHACL features . . . 120
3.1.5 Generating sequential IDs with the Sequences plugin . . . 122
3.1.5.1 What the plugin does . . . 122
3.1.5.2 Usage . . . 122
3.1.6 Loading via HTTP with curl . . . 125
3.2 Exploring your data . . . 126
3.3 SPARQL queries . . . 127
3.3.1 Save and share queries . . . 129
3.3.2 Interrupt queries . . . 130
3.4 SPARQL plugins . . . 130
3.4.1 Ranking results . . . 130
3.4.1.1 Ranking entities by interconnectedness with the RDF Rank Plugin . . . 130
3.4.1.2 Prominence . . . 135
3.4.2 Geospatial extensions . . . 135
3.4.2.1 What are geospatial extensions . . . 135
3.4.2.2 Creating a geospatial index . . . 136
3.4.2.3 Geospatial query syntax . . . 137
3.4.2.4 Extension query functions . . . 139
3.4.2.5 Implementation details . . . 140
3.4.3 GeoSPARQL support . . . 141
3.4.3.1 What is GeoSPARQL . . . 141
3.4.3.2 Usage . . . 142
3.4.3.3 GeoSPARQL examples . . . 146
3.4.4 Full-text search . . . 151
3.4.4.1 Full-text search using the GraphDB connectors . . . 151
3.4.4.2 Simple full-text search index . . . 151
3.4.5 SPARQL Federation . . . 165
3.4.5.1 Internal SPARQL federation . . . 166
3.4.5.2 Federated query to a remote password-protected repository . . . 168
3.4.6 The SPARQL-MM multimedia extension . . . 169
3.4.6.1 Usage examples . . . 169
3.4.7 Data history and versioning . . . 171
3.4.7.1 Usage . . . 171
3.4.7.2 Query process and examples . . . 174
3.4.7.3 Index components . . . 178
3.4.8 SQL access over JDBC . . . 178
3.4.8.1 Configuration . . . 179
3.4.8.2 Type mapping . . . 182
3.4.8.3 WHERE to FILTER conversion . . . 183
3.4.8.4 Table verification . . . 183
3.4.8.5 Usage examples . . . 185
3.4.8.6 How it works: Table description . . . 189
3.4.9 Query profiling with the Explain plan . . . 190
3.4.9.1 Activating the Explain plan . . . 190
3.4.9.2 A simple Explain plan . . . 190
3.4.9.3 Multiple triple patterns . . . 191
3.4.9.4 Sample wine queries . . . 192
3.4.10 Change tracking . . . 194
3.4.10.1 What the plugin does . . . 194
3.4.10.2 Usage . . . 194
3.5 Visualize and explore . . . 196
3.5.1 Class hierarchy . . . 196
3.5.1.1 Explore your data — different actions . . . 197
3.5.1.2 Domain-range graph . . . 199
3.5.2 Class relationships . . . 200
3.5.3 Explore resources . . . 203

3.5.3.1 Explore resources through the easy graph . . . 203
3.5.3.2 Create your own visual graph . . . 208
3.5.3.3 Save and share visual graphs . . . 209
3.5.3.4 Embed visual graphs . . . 210
3.5.4 View and edit resources . . . 210
3.5.4.1 View and add a resource . . . 210
3.5.4.2 Edit a resource . . . 211
3.6 GraphQL . . . 212
3.6.1 GraphQL security . . . 212
3.6.2 GraphQL endpoints . . . 212
3.6.3 Licensing GraphQL . . . 213
3.6.4 GraphQL Playground . . . 213
3.6.4.1 Using your GraphQL endpoints with the GraphiQL Explorer . . . 214
3.6.5 Generating an endpoint from SHACL using the Star Wars dataset . . . 220
3.6.5.1 Prerequisites . . . 220
3.6.5.2 Generate an endpoint from named graph . . . 220
3.6.5.3 Executing simple queries in the GraphQL Playground . . . 221
3.7 GraphQL schemas and endpoint management . . . 223
3.7.1 Creating GraphQL endpoints . . . 223
3.7.1.1 Creating a GraphQL interface from GraphQL schema shapes . . . 224
3.7.1.2 Creating a GraphQL interface from OWL ontologies and SHACL shapes . . . 231
3.7.2 GraphQL schema generation configuration options . . . 233
3.7.3 Managing GraphQL endpoints . . . 234
3.7.3.1 Endpoint configuration settings . . . 236
3.7.4 Using language specifications in your GraphQL queries . . . 237
3.7.5 Using and managing GraphQL schemas with the GraphDB REST API . . . 238
3.7.6 Role-Based Access Control . . . 239
3.7.6.1 Role Definitions (Objects and Schema) . . . 240
3.7.6.2 Actions and notActions . . . 240
3.7.6.3 RBAC Filters . . . 241
3.8 Semantic similarity searches . . . 242
3.8.1 What can the similarity plugin do for me? . . . 242
3.8.2 What does the similarity plugin do? . . . 243
3.8.3 How does the similarity plugin work? . . . 243
3.8.4 Download sample data . . . 243
3.8.5 Text-based similarity searches . . . 244
3.8.5.1 Create text similarity index . . . 244
3.8.5.2 Use the index to search for similar terms . . . 245
3.8.5.3 Search documents for which selected term is specific . . . 245
3.8.5.4 Search specific terms in selected document . . . 246
3.8.5.5 Search for closest documents . . . 247
3.8.5.6 Search similar news within a specified time range . . . 247
3.8.5.7 Term to term search . . . 248
3.8.6 Customizing index creation . . . 250
3.8.6.1 Create index parameters . . . 250
3.8.6.2 Disabled parameters . . . 251
3.8.6.3 Stop words and Lucene analyzer . . . 251
3.8.6.4 Search parameters . . . 251
3.8.6.5 Delete an index using a SPARQL query . . . 252
3.8.6.6 Rebuild an index using a SPARQL query . . . 252
3.8.6.7 Search in the index during rebuild with no downtime . . . 252
3.8.6.8 Locality-sensitive hashing . . . 252
3.8.6.9 Boosting a term's weight . . . 254
3.8.7 Predication-based Semantic Indexing . . . 254
3.8.7.1 Load example data . . . 254
3.8.7.2 Create a predication-based index . . . 254
3.8.7.3 Search predication-based index . . . 256
3.8.7.4 Analogical searches . . . 257

3.8.7.5 Why is this important? . . . 257
3.8.8 Hybrid indexing . . . 258
3.8.8.1 Edit the FactForge data . . . 258
3.8.8.2 Create a Literal index . . . 259
3.8.8.3 Use the Literal index . . . 259
3.8.8.4 Indexing behavior . . . 260
3.8.8.5 Manual creation . . . 260
3.8.9 Using training cycles . . . 260
3.9 Querying GPT models . . . 261
3.9.1 Prerequisites and additional configurations . . . 261
3.9.2 GPT functions . . . 261
3.9.2.1 Setting the temperature of the response . . . 261
3.9.2.2 `gpt:ask()` — Retrieve a single answer . . . 262
3.9.2.3 `gpt:list()` — Retrieve a list of answers . . . 263
3.9.2.4 `gpt:table()` — Retrieve a table of answers . . . 264
3.9.3 List manipulation extension functions . . . 265
3.9.3.1 `helper:tuple()` — Combine values into a list . . . 265
3.9.3.2 `helper:tupleAggr()` — Aggregate values into a list . . . 266
3.9.3.3 `helper:rdf()` — Combine values into an RDF triple . . . 267
3.9.3.4 `helper:serializeRDF()` — Convert internal RDF to readable triples . . . 267
3.9.3.5 `helper:iterate()` — Iterate through an internal list . . . 268
3.9.4 GPT query explanations . . . 268
3.9.4.1 Customizing the explanation output . . . 269
3.9.4.2 Setting the temperature of the explanation . . . 271
3.10 Talk to Your Graph . . . 271
3.10.1 Prerequisites and configuration . . . 272
3.10.1.1 Model Requirements . . . 272
3.10.2 Talk to Your Graph query methods . . . 272
3.10.2.1 IRI discovery query methods . . . 274
3.10.3 Create and configure a Talk to Your Graph agent . . . 274
3.10.3.1 Talk to Your Graph agent configuration parameters . . . 274
3.10.3.2 Manage Talk to Your Graph agents . . . 276
3.10.4 Using the Talk to Your Graph chat interface . . . 277
3.10.4.1 Explaining agent responses . . . 278
3.10.4.2 Managing chats . . . 278
3.11 Graph path search . . . 279
3.11.1 Overview . . . 279
3.11.2 Usage . . . 279
3.11.2.1 Search algorithms . . . 282
3.11.2.2 Search modifiers . . . 283
3.11.3 Usage examples . . . 284
3.11.3.1 Shortest path . . . 284
3.11.3.2 All paths . . . 287
3.11.3.3 Shortest distance . . . 293
3.11.3.4 Cyclic path . . . 293
3.11.3.5 Parallel search mode . . . 294
3.11.3.6 Exportable graph pattern bindings . . . 295
3.11.3.7 Bidirectional search . . . 296
3.12 Reasoning . . . 298
3.12.1 Logical formalism . . . 298
3.12.2 Rule format and semantics . . . 298
3.12.3 The ruleset file . . . 299
3.12.3.1 Prefixes . . . 299
3.12.3.2 Axioms . . . 299
3.12.3.3 Rules . . . 300
3.12.4 Rulesets . . . 304
3.12.4.1 Predefined rulesets . . . 304
3.12.4.2 Custom rulesets . . . 305

3.12.5 Inference . . . 305
3.12.5.1 Compiling the reasoner . . . 306
3.12.5.2 Rulesets execution . . . 306
3.12.5.3 Retraction of assertions . . . 306
3.12.6 Available ruleset operations . . . 308
3.12.6.1 Operations on rulesets . . . 308
3.12.6.2 Reinferring . . . 311
3.12.7 Provenance . . . 311
3.13 Exporting data . . . 311
3.13.1 JSON-LD and NDJSON-LD export forms . . . 311
3.13.2 Exporting a repository . . . 312
3.13.3 Exporting individual graphs . . . 312
3.13.4 Exporting query results . . . 313
3.13.5 Exporting resources . . . 313
3.13.6 Exporting via HTTP with curl . . . 313
3.13.6.1 Request headers . . . 313
3.13.6.2 Fetch statements from repository . . . 314
3.13.6.3 Fetch queries from repository . . . 314
3.13.7 Configuring the JSON-LD writer properties . . . 315

4 Connectors **317**
4.1 Connecting to external components and services . . . 317
4.1.1 Full-text search and aggregation connectors . . . 317
4.1.2 ChatGPT Retrieval connector . . . 318
4.1.3 MongoDB integration . . . 318
4.1.4 Kafka connector . . . 318
4.2 General full-text search with the Connectors . . . 319
4.2.1 Indexing with wildcard literals . . . 319
4.2.2 Deriving field names from a predicate URI . . . 319
4.2.3 Any type or untyped indexing . . . 319
4.2.4 Examples . . . 319
4.2.4.1 Indexing all literals . . . 320
4.2.4.2 Indexing all literals in distinct fields . . . 321
4.3 Lucene GraphDB Connector . . . 322
4.3.1 Usage . . . 323
4.3.2 Setup and maintenance . . . 324
4.3.2.1 Creating a connector instance . . . 324
4.3.2.2 Dropping a connector instance . . . 327
4.3.2.3 Retrieving the create options for a connector instance . . . 327
4.3.2.4 Listing available connector instances . . . 327
4.3.2.5 Instance status check . . . 328
4.3.3 Working with data . . . 329
4.3.3.1 Adding, updating, and deleting data . . . 329
4.3.3.2 Simple queries . . . 329
4.3.3.3 Combining Lucene results with GraphDB data . . . 329
4.3.3.4 Entity match score . . . 330
4.3.3.5 Basic facet queries . . . 330
4.3.3.6 Sorting . . . 331
4.3.3.7 Limit and offset . . . 332
4.3.3.8 Snippet extraction . . . 332
4.3.3.9 Total hits . . . 333
4.3.4 List of creation parameters . . . 333
4.3.4.1 Special field definitions . . . 337
4.3.5 Datatype mapping . . . 341
4.3.5.1 Date and time conversion . . . 342
4.3.6 Entity filtering . . . 342
4.3.6.1 Types of filters . . . 343
4.3.6.2 Filter operators . . . 343

4.3.6.3 Filter modifiers . . . 345
4.3.6.4 Two-variable filtering . . . 346
4.3.6.5 Basic entity filter example . . . 346
4.3.6.6 Advanced entity filter example . . . 347
4.3.7 Overview of connector predicates . . . 350
4.3.8 Caveats . . . 350
4.3.8.1 Order of control . . . 350
4.3.9 Upgrading from previous versions . . . 350
4.3.9.1 Migrating from GraphDB 9.x . . . 350
4.4 Solr GraphDB Connector . . . 351
4.4.1 Usage . . . 352
4.4.2 Setup and maintenance . . . 353
4.4.2.1 Prerequisites . . . 353
4.4.2.2 Creating a connector instance . . . 354
4.4.2.3 Dropping a connector instance . . . 359
4.4.2.4 Retrieving the create options for a connector instance . . . 359
4.4.2.5 Listing available connector instances . . . 359
4.4.2.6 Instance status check . . . 360
4.4.3 Working with data . . . 361
4.4.3.1 Adding, updating, and deleting data . . . 361
4.4.3.2 Simple queries . . . 361
4.4.3.3 Raw queries . . . 361
4.4.3.4 Basic facet queries . . . 363
4.4.3.5 Advanced facet and aggregation queries . . . 364
4.4.3.6 Sorting . . . 365
4.4.3.7 Limit and offset . . . 365
4.4.3.8 Snippet extraction . . . 366
4.4.3.9 Total hits . . . 367
4.4.4 List of creation parameters . . . 367
4.4.4.1 Updating parameters at runtime . . . 370
4.4.4.2 Special field definitions . . . 371
4.4.5 Datatype mapping . . . 375
4.4.5.1 Date and time conversion . . . 376
4.4.6 Entity filtering . . . 376
4.4.6.1 Types of filters . . . 377
4.4.6.2 Filter operators . . . 377
4.4.6.3 Filter modifiers . . . 379
4.4.6.4 Two-variable filtering . . . 380
4.4.6.5 Basic entity filter example . . . 380
4.4.6.6 Advanced entity filter example . . . 381
4.4.7 Overview of connector predicates . . . 384
4.4.8 SolrCloud support . . . 385
4.4.8.1 Zookeeper instances . . . 385
4.4.8.2 SolrCloud collection configsets . . . 385
4.4.9 Caveats . . . 386
4.4.9.1 Order of control . . . 386
4.4.10 Upgrading from previous versions . . . 387
4.4.10.1 Migrating from GraphDB 9.x . . . 387
4.5 Elasticsearch GraphDB Connector . . . 387
4.5.1 Usage . . . 388
4.5.2 Setup and maintenance . . . 390
4.5.2.1 Prerequisites . . . 390
4.5.2.2 Creating a connector instance . . . 390
4.5.2.3 Dropping a connector instance . . . 394
4.5.2.4 Retrieving the create options for a connector instance . . . 394
4.5.2.5 Listing available connector instances . . . 394
4.5.2.6 Instance status check . . . 395
4.5.3 Working with data . . . 396

4.5.3.1 Adding, updating and deleting data . . . 396
4.5.3.2 Simple queries . . . 396
4.5.3.3 Raw queries . . . 396
4.5.3.4 Vector search queries . . . 398
4.5.3.5 Basic facet queries . . . 400
4.5.3.6 Advanced facet and aggregation queries . . . 401
4.5.3.7 Sorting . . . 402
4.5.3.8 Limit and offset . . . 402
4.5.3.9 Snippet extraction . . . 403
4.5.3.10 Total hits . . . 405
4.5.4 List of creation parameters . . . 405
4.5.4.1 Updating parameters at runtime . . . 410
4.5.4.2 Special field definitions . . . 410
4.5.5 Datatype mapping . . . 419
4.5.5.1 Date and time conversion . . . 420
4.5.6 Entity filtering . . . 420
4.5.6.1 Types of filters . . . 420
4.5.6.2 Filter operators . . . 421
4.5.6.3 Filter modifiers . . . 423
4.5.6.4 Two-variable filtering . . . 424
4.5.6.5 Basic entity filter example . . . 424
4.5.6.6 Advanced entity filter example . . . 425
4.5.6.7 Migrating filters from GraphDB 9.x . . . 428
4.5.7 Overview of connector predicates . . . 428
4.5.8 Caveats . . . 429
4.5.8.1 Order of control . . . 429
4.5.9 Upgrading from previous versions . . . 429
4.5.9.1 Migrating from GraphDB 9.x . . . 429
4.6 OpenSearch GraphDB Connector . . . 430
4.6.1 Usage . . . 431
4.6.2 Setup and maintenance . . . 433
4.6.2.1 Prerequisites . . . 433
4.6.2.2 Creating a connector instance . . . 433
4.6.2.3 Dropping a connector instance . . . 436
4.6.2.4 Retrieving the create options for a connector instance . . . 437
4.6.2.5 Listing available connector instances . . . 437
4.6.2.6 Instance status check . . . 438
4.6.3 Working with data . . . 438
4.6.3.1 Adding, updating and deleting data . . . 438
4.6.3.2 Simple queries . . . 438
4.6.3.3 Raw queries . . . 439
4.6.3.4 Vector search queries . . . 441
4.6.3.5 Basic facet queries . . . 442
4.6.3.6 Advanced facet and aggregation queries . . . 443
4.6.3.7 Sorting . . . 444
4.6.3.8 Limit and offset . . . 444
4.6.3.9 Snippet extraction . . . 445
4.6.3.10 Total hits . . . 447
4.6.4 List of creation parameters . . . 447
4.6.4.1 Updating parameters at runtime . . . 451
4.6.4.2 Special field definitions . . . 452
4.6.5 Datatype mapping . . . 461
4.6.5.1 Date and time conversion . . . 461
4.6.6 Entity filtering . . . 462
4.6.6.1 Types of filters . . . 462
4.6.6.2 Filter operators . . . 463
4.6.6.3 Filter modifiers . . . 465
4.6.6.4 Two-variable filtering . . . 466

4.6.6.5 Basic entity filter example . . . 466
4.6.6.6 Advanced entity filter example . . . 467
4.6.6.7 Migrating filters from GraphDB 9.x . . . 470
4.6.7 Overview of connector predicates . . . 470
4.6.8 Caveats . . . 471
4.6.8.1 Order of control . . . 471
4.6.9 Upgrading from previous versions . . . 471
4.6.9.1 Migrating from GraphDB 9.x . . . 471
4.6.10 Migrating from Elasticsearch to OpenSearch Connector . . . 472
4.7 Kafka GraphDB Connector . . . 472
4.7.1 Usage . . . 473
4.7.2 Setup and maintenance . . . 475
4.7.2.1 Prerequisites . . . 475
4.7.2.2 Creating a connector instance . . . 475
4.7.2.3 Dropping a connector instance . . . 477
4.7.2.4 Retrieving the create options for a connector instance . . . 477
4.7.2.5 Listing available connector instances . . . 477
4.7.2.6 Instance status check . . . 478
4.7.3 Working with data . . . 478
4.7.3.1 Adding, updating, and deleting data . . . 478
4.7.4 List of creation parameters . . . 478
4.7.4.1 Updating parameters at runtime . . . 482
4.7.4.2 Special field definitions . . . 483
4.7.5 Datatype mapping . . . 491
4.7.5.1 Date and time conversion . . . 491
4.7.6 Entity filtering . . . 492
4.7.6.1 Types of filters . . . 492
4.7.6.2 Filter operators . . . 493
4.7.6.3 Filter modifiers . . . 495
4.7.6.4 Two-variable filtering . . . 496
4.7.6.5 Basic entity filter example . . . 496
4.7.6.6 Advanced entity filter example . . . 497
4.7.7 Overview of connector predicates . . . 499
4.7.8 Caveats . . . 500
4.7.8.1 Producer sharing . . . 500
4.7.9 Upgrading from previous versions . . . 501
4.7.9.1 Migrating from GraphDB 9.x . . . 501
4.8 Kafka Sink Connector . . . 501
4.8.1 Setup . . . 502
4.8.2 Update types . . . 502
4.8.2.1 Simple add . . . 502
4.8.2.2 Replace graph . . . 504
4.8.2.3 DELETE/INSERT template . . . 505
4.8.3 Configuration properties . . . 508
4.9 ChatGPT Retrieval GraphDB Connector . . . 510
4.9.1 Usage . . . 510
4.9.2 Setup and maintenance . . . 512
4.9.2.1 Setting up the ChatGPT retrieval plugin . . . 512
4.9.2.2 Creating a connector instance . . . 512
4.9.2.3 Dropping a connector instance . . . 517
4.9.2.4 Retrieving the create options for a connector instance . . . 518
4.9.2.5 Listing available connector instances . . . 518
4.9.2.6 Instance status check . . . 519
4.9.3 Working with data . . . 519
4.9.3.1 Adding, updating and deleting data . . . 519
4.9.3.2 Simple queries . . . 519
4.9.3.3 Raw queries . . . 520
4.9.3.4 Limit . . . 521

4.9.3.5 Chunk and metadata extraction . . . 522
4.9.4 List of creation parameters . . . 523
4.9.4.1 Updating parameters at runtime . . . 525
4.9.4.2 Special field definitions . . . 526
4.9.5 Entity filtering . . . 530
4.9.5.1 Types of filters . . . 530
4.9.5.2 Filter operators . . . 531
4.9.5.3 Filter modifiers . . . 533
4.9.5.4 Two-variable filtering . . . 534
4.9.5.5 Basic entity filter example . . . 534
4.9.5.6 Advanced entity filter example . . . 535
4.9.6 Text document assembly . . . 537
4.9.7 Overview of connector predicates . . . 538
4.9.8 Caveats . . . 538
4.9.8.1 Order of control . . . 538
4.10 MongoDB integration . . . 538
4.10.1 Usage . . . 539
4.10.2 Setup and maintenance . . . 542
4.10.2.1 Installing MongoDB . . . 542
4.10.2.2 Creating an index . . . 542
4.10.2.3 Upgrading an index . . . 542
4.10.2.4 Deleting an index . . . 543
4.10.2.5 Loading sample data . . . 543
4.10.2.6 Querying MongoDB . . . 543
4.10.2.7 Batch processing . . . 544
4.10.2.8 Multiple index calls in the same query . . . 545
4.10.2.9 Using aggregation functions . . . 546
4.10.2.10 Custom fields . . . 547
4.10.2.11 Authentication . . . 547
4.11 Text mining plugin . . . 548
4.11.1 Usage examples . . . 548
4.11.1.1 spaCy server . . . 548
4.11.1.2 GATE Cloud . . . 551
4.11.1.3 Tag . . . 552
4.11.1.4 Extracting annotations from another NER service . . . 557
4.11.1.5 Comparing annotations between services . . . 564
4.11.1.6 Enriching documents with mentions of known entities . . . 565
4.11.2 Error handling . . . 566
4.11.3 Managing text mining instances . . . 567
4.11.3.1 Listing all clients . . . 567
4.11.3.2 Getting a client's configuration . . . 568
4.11.3.3 Dropping an instance . . . 568
4.11.4 Monitoring annotation progress . . . 568
4.12 Embedding model configuration for vector search . . . 568
4.12.1 Using a local embedding model deployed with Graphwise Transformer . . . 569
4.12.2 Using an OpenAI embedding model . . . 569
4.12.3 Using an alternative model provider through the EmbeddingModel interface of langchain4j 569

5 Clusters **571**
5.1 Overview of clusters . . . 571
5.1.1 The Raft consensus algorithm . . . 571
5.1.1.1 Cluster roles . . . 572
5.1.2 Internal and external proxy . . . 572
5.1.2.1 Internal proxy . . . 572
5.1.2.2 External proxy . . . 572
5.1.3 High availability and fault-tolerance . . . 573
5.1.3.1 What is consensus? . . . 573
5.1.3.2 High availability deployment . . . 573

5.1.4 Quorum-based replication . . . 574
5.1.4.1 Query load balancer . . . 574
5.1.4.2 Local consistency . . . 574
5.2 Creating and managing a cluster . . . 575
5.2.1 Prerequisites . . . 575
5.2.1.1 Ports . . . 575
5.2.1.2 Configuration properties prerequisites . . . 576
5.2.1.3 Achieving a high availability deployment . . . 577
5.2.2 Create cluster . . . 577
5.2.2.1 Using the Workbench . . . 577
5.2.2.2 Using the REST API . . . 578
5.2.2.3 Creation parameters . . . 580
5.2.3 Manage cluster membership . . . 580
5.2.3.1 Managing cluster membership through Workbench . . . 581
5.2.3.2 Managing cluster membership using the REST API . . . 582
5.2.4 Manage cluster configuration properties . . . 584
5.2.4.1 Using the Workbench . . . 584
5.2.4.2 Using the REST API . . . 585
5.2.5 Monitor cluster status . . . 586
5.2.5.1 Cluster group . . . 587
5.2.5.2 Cluster node . . . 588
5.2.6 Delete cluster . . . 589
5.2.6.1 Force parameter . . . 589
5.2.7 Configure external cluster proxy . . . 590
5.2.7.1 Start the external proxy . . . 590
5.2.7.2 External proxy for cluster running over SSL . . . 591
5.2.8 Cluster security . . . 591
5.2.8.1 Encryption . . . 591
5.2.8.2 Access control . . . 592
5.2.9 Truncate cluster transaction log . . . 592
5.2.10 Disaster recovery . . . 592
5.2.10.1 Recovering individual nodes . . . 592
5.2.10.2 Rebuilding the cluster from a single node . . . 593
5.3 Cluster states and election procedures . . . 594
5.3.1 Leader election . . . 594
5.3.2 Terms . . . 595
5.3.3 Cluster states . . . 595
5.3.3.1 Cluster node states . . . 595
5.3.3.2 Cluster sync states . . . 596
5.3.3.3 Cluster link states . . . 596
5.3.4 Transaction fingerprints . . . 596
5.3.5 Log replication . . . 597
5.4 Using a cluster . . . 598
5.4.1 Using the GraphDB client API for Java . . . 599
5.4.2 Using a cluster with an external proxy . . . 600
5.4.3 Setting local consistency . . . 600
5.4.3.1 Using the GraphDB Java client API . . . 601
5.5 Multi-region cluster deployments . . . 601
5.5.1 Architecture . . . 602
5.5.2 Secondary cluster identifier tags . . . 603
5.5.2.1 Adding identifier tags using Workbench . . . 603
5.5.2.2 Adding identifier tags using the REST API . . . 603
5.5.3 Enable the secondary cluster mode . . . 604
5.5.3.1 Prerequisites . . . 604
5.5.3.2 Enable the secondary cluster mode using Workbench . . . 604
5.5.3.3 Enable the secondary cluster mode using the REST API . . . 605
5.5.4 Monitoring multi-region cluster deployments . . . 605
5.5.4.1 Monitoring a multi-region cluster deployment using Workbench . . . 606

5.5.4.2 Monitoring a multi-region cluster deployment using the REST API 607

6 Administration **609**
6.1 Setting up your license 609
6.1.1 Setting up your license through the Workbench 609
6.1.2 Setting up your license through a configuration file 610
6.1.2.1 Custom file path property 610
6.1.3 Where GraphDB looks for a license 610
6.2 Security 610
6.2.1 Enabling security 610
6.2.1.1 Login and default credentials 611
6.2.1.2 Free access 611
6.2.2 User management 613
6.2.2.1 Create new user 613
6.2.2.2 Delete user 614
6.2.2.3 Edit user account details 614
6.2.3 Access control 615
6.2.3.1 Authorization and user database 615
6.2.3.2 Authentication methods 623
6.2.3.3 Example configurations 633
6.2.4 Fine-grained access control 639
6.2.4.1 How FGAC works 640
6.2.4.2 Rule scopes for Fine-grained access control 641
6.2.4.3 Managing FGAC rules through Workbench 644
6.2.4.4 Managing FGAC rules through the REST API 646
6.2.4.5 Fine-grained access control tutorials 649
6.2.5 Custom user roles 660
6.2.5.1 Mapping OAuth custom roles 660
6.2.5.2 Managing custom roles through Workbench 660
6.2.5.3 Custom role management REST API 661
6.2.6 Encryption 663
6.2.6.1 Encryption in transit 663
6.2.6.2 Encryption at rest 665
6.2.7 Security auditing 665
6.3 Backup and restore 666
6.3.1 Planning a backup 667
6.3.1.1 Planning cloud backups 668
6.3.2 Creating a backup 670
6.3.2.1 Backup options 670
6.3.2.2 Full data backup 671
6.3.2.3 Partial data backup 671
6.3.2.4 Full data and system backup 671
6.3.2.5 Backup of only system data 672
6.3.2.6 Creating a cloud backup 672
6.3.3 Restoring from a backup 673
6.3.3.1 Restore options 673
6.3.3.2 Full data restore preserving other repositories 674
6.3.3.3 Full data restore with replace 674
6.3.3.4 Partial data restore 674
6.3.3.5 Restoring only system data 674
6.3.3.6 Restoring from a cloud backup 675
6.3.4 Backup compatibility between different versions of GraphDB 676
6.3.4.1 Backup-restore compatibility between different patch versions 676
6.3.4.2 Backup-restore compatibility between different minor versions 676
6.3.4.3 Backup-restore compatibility between different major versions 676
6.3.5 Monitoring your recovery operations 676
6.3.5.1 Monitoring your recovery operations through cURL 677
6.4 Monitoring and troubleshooting 677

6.4.1 Request tracking . . . 677
6.4.2 Monitoring your AWS deployment . . . 678
6.4.3 Monitoring your Azure deployment . . . 679
6.4.3.1 Configuring the monitoring for GraphDB in Azure . . . 679
6.4.3.2 Using Azure Application Insights to monitor your GraphDB deployment . . . 681
6.4.4 Repository health checks . . . 684
6.4.4.1 Possible values for health checks and their meaning . . . 684
6.4.4.2 Aggregated health checks . . . 685
6.4.4.3 Passive health checks . . . 686
6.4.5 Low disk space health checks . . . 687
6.4.5.1 Recovering from a fatal state . . . 687
6.4.6 System monitoring . . . 688
6.4.6.1 Workbench monitoring . . . 688
6.4.6.2 Prometheus monitoring . . . 689
6.4.6.3 JMX console monitoring . . . 692
6.4.7 Diagnosing and reporting critical errors . . . 693
6.4.7.1 Running a system report . . . 693
6.4.7.2 Logs . . . 695
6.5 Docker and Helm . . . 696
6.6 Configuring GraphDB memory . . . 696
6.6.1 Configure Java heap memory . . . 696
6.6.2 Single global page cache . . . 697
6.6.3 Configure entity pool memory . . . 697
6.6.4 Sample memory configuration . . . 698
6.6.5 Setting a tupleset/distinct memory threshold . . . 699
6.7 Performance optimizations . . . 699
6.7.1 Data loading and index optimizations . . . 699
6.7.1.1 Dataset loading . . . 699
6.7.1.2 Optional GraphDB indexes . . . 701
6.7.1.3 Cache and index monitoring and optimizations . . . 701
6.7.1.4 Index compacting . . . 702
6.7.2 Query optimizations . . . 703
6.7.2.1 Caching literal language tags . . . 703
6.7.2.2 Not enumerating sameAs . . . 703
6.7.3 Inference optimizations . . . 704
6.7.3.1 Delete optimizations . . . 704
6.7.3.2 Rules optimizations . . . 706
6.7.3.3 Optimization of `owl:sameAs` . . . 716
6.7.3.4 RDFS and OWL support optimizations . . . 722
6.8 Administration via HTTP with curl . . . 722
6.8.1 Cluster group management . . . 723
6.8.2 Cluster monitoring . . . 724
6.8.3 Infrastructure monitoring . . . 725
6.8.4 Location management . . . 725
6.8.5 Repository SHACL validation . . . 726
6.8.6 Saved queries . . . 727
6.8.7 Security management . . . 727
6.8.8 SPARQL template management . . . 729
6.8.9 SQL views management . . . 730
6.8.10 Authentication . . . 731
6.8.11 Structures monitoring . . . 732
6.9 Migrating GraphDB configurations . . . 732
6.9.1 Compatibility between the versions of GraphDB, Connectors, and third-party connectors 732
6.9.2 Migrating GraphDB . . . 736
6.9.3 Migrating plugins in a cluster . . . 737
6.10 Migrating GraphQL configurations from Semantic Objects to GraphDB . . . 737
6.10.1 General notes . . . 737
6.10.2 Licence changes . . . 738

6.10.3 Deployment migration . . . 738
6.10.4 API Calls . . . 738
6.10.5 Configuration parameters . . . 739
6.10.6 Schema changes . . . 739
6.10.6.1 Schema migration . . . 740
6.10.7 RBAC migration . . . 741
6.11 How to deploy GraphDB in AWS . . . 741
6.11.1 Architecture . . . 741
6.11.2 Prerequisites . . . 742
6.11.2.1 Required skills . . . 743
6.11.2.2 AWS billable services required by GraphDB . . . 743
6.11.3 Setting up your Virtual Private Cloud (VPC) . . . 744
6.11.4 Setting up your Route 53 private hosted zone . . . 745
6.11.5 Creating an S3 bucket . . . 746
6.11.6 Importing a certificate into Amazon Certificate Manager (ACM) . . . 746
6.11.7 Setting up the Load Balancer . . . 747
6.11.8 Setting up instance role and profile permissions . . . 748
6.11.8.1 Allow the EC2 instance to read, write, and list objects in S3 (optional) . . . 748
6.11.8.2 Allow the listing of EC2 instances . . . 749
6.11.8.3 Allow the listing, creating, attaching, and tagging of EBS volumes . . . 749
6.11.8.4 Allow adding of records to Route 53 private hosted zone . . . 750
6.11.9 Creating an IAM role and attaching policies . . . 751
6.11.10 Launching GraphDB instances . . . 751
6.11.10.1 Setting up the launch template . . . 751
6.11.10.2 Creating a cluster . . . 753
6.11.10.3 Opening GraphDB instances . . . 753
6.11.11 Updating GraphDB configurations and versions . . . 753
6.11.11.1 Stopping individual EC2 instances . . . 754
6.11.11.2 Scaling the cluster out and then back in . . . 754
6.12 How to deploy GraphDB in Azure . . . 754
6.12.1 Architecture . . . 755
6.12.2 Prerequisites . . . 755
6.12.2.1 Technical requirements . . . 756
6.12.2.2 Required skills . . . 756
6.12.3 Creating a Resource Group . . . 757
6.12.4 Creating a Virtual Network . . . 758
6.12.4.1 Creating the two subnets . . . 758
6.12.5 Creating an Application Security Groups (ASG) . . . 759
6.12.6 Creating the two Network Security Groups (NSG) . . . 759
6.12.6.1 Configuring the security rules for the VMSS NSG . . . 760
6.12.6.2 Configuring the security rules for the Gateway NSG . . . 761
6.12.7 Creating the two Public IP addresses . . . 761
6.12.8 Creating the NAT gateway . . . 762
6.12.9 Creating a User-Assigned Managed Identity . . . 762
6.12.10 Creating the Storage Account . . . 762
6.12.11 Creating the Application Gateway . . . 763
6.12.12 Creating the Virtual Machine Scale Set (VMSS) . . . 765
6.12.12.1 Attaching the managed disks . . . 767
6.12.13 Setting up and managing your cluster . . . 767
6.12.14 Accessing the Workbench through the Application Gateway . . . 767
6.12.15 Scaling out and back in VMSS . . . 767
6.12.16 Required manual maintenance of instances . . . 768
6.13 How to deploy GraphDB in GCP . . . 768
6.13.1 Architecture . . . 768
6.13.2 Prerequisites . . . 769
6.13.2.1 Technical requirements . . . 769
6.13.2.2 Required skills . . . 769
6.13.3 Setting up your Virtual Private Cloud (VPC) . . . 770

6.13.4 Setting up your Cloud DNS private hosted zone . . . 771
6.13.5 Creating a Cloud Storage Bucket . . . 771
6.13.6 Importing a certificate into GCP Certificate Manager . . . 772
6.13.7 Setting up the Load Balancer . . . 772
6.13.8 Setting up service account . . . 773
6.13.9 Launching GraphDB instances . . . 773
6.13.9.1 Setting up Instance Group . . . 774
6.13.9.2 Creating a cluster . . . 774
6.13.9.3 Opening GraphDB instances . . . 774
6.13.10 Updating GraphDB configurations and versions . . . 774
6.13.10.1 Stopping individual GCE instances . . . 775
6.13.10.2 Scaling the cluster out and then back in . . . 775
6.14 Using GraphDB's LLM tools with external clients . . . 775
6.14.1 Using GraphDB's LLM tools with MCP clients . . . 776
6.14.1.1 MCP security configurations . . . 776
6.14.1.2 Exposed LLM tools for MCP clients . . . 776
6.14.1.3 SSE transport and the MCP gateway . . . 777
6.14.2 Using GraphDB's LLM tools with Dify and other LLM agent frameworks . . . 777
6.14.2.1 Supported configuration types . . . 777
6.14.2.2 Exposed tools for the Dify framework . . . 777
6.14.2.3 Example yaml configuration . . . 778
6.14.2.4 Using Dify API extension . . . 779

7 Tutorials **781**
7.1 Running the interactive user guides . . . 781
7.1.1 Available guides . . . 781
7.1.2 Running a guide . . . 782
7.2 GraphDB Fundamentals Video Series . . . 782
7.2.1 Module 1: RDF & RDFS . . . 782
7.2.2 Module 2: SPARQL . . . 783
7.2.3 Module 3: Ontology . . . 783
7.2.4 Module 4: GraphDB Installation . . . 783
7.2.5 Module 5: GraphDB Workbench & REST API . . . 783
7.2.6 Module 6: Loading Data . . . 783
7.2.7 Module 7: Rulesets & Reasoning Strategies . . . 783
7.2.8 Module 8: Virtualization . . . 783
7.2.9 Module 9: Plugins . . . 784
7.2.10 Module 10: Connectors . . . 784
7.3 Loading and querying data . . . 784
7.3.1 Load your data . . . 784
7.3.1.1 Load data through the GraphDB Workbench . . . 784
7.3.1.2 Load data through SPARQL or the RDF4J API . . . 785
7.3.1.3 Load data through the ImportRDF tool . . . 786
7.3.2 Explore data and class relationships . . . 786
7.3.2.1 Explore instances . . . 786
7.3.2.2 Create your own visual graph . . . 788
7.3.2.3 Class hierarchy . . . 789
7.3.2.4 Domain-Range graph . . . 792
7.3.2.5 Class relationships . . . 793
7.3.3 Query your data . . . 796
7.3.3.1 Query data through the Workbench . . . 796
7.3.3.2 Query data programmatically . . . 801
7.3.4 Visualize GraphDB data with Ogma JS . . . 801
7.3.4.1 People and organizations related to Google in factforge.net . . . 802
7.3.4.2 Suspicious control chain through off-shore companies in factforge.net . . . 805
7.3.4.3 Shortest flight path . . . 808
7.3.4.4 Common function to visualize GraphDB data . . . 813
7.4 GraphDB and W3C standards . . . 814

7.4.1 Introduction to the Semantic Web . . . 814
7.4.1.1 Resource Description Framework (RDF) . . . 814
7.4.1.2 RDF Schema (RDFS) . . . 818
7.4.1.3 Ontologies and knowledge bases . . . 821
7.4.1.4 Logic and inference . . . 823
7.4.1.5 The Web Ontology Language (OWL) and its dialects . . . 825
7.4.1.6 Query languages . . . 828
7.4.1.7 Reasoning strategies . . . 829
7.4.1.8 Semantic repositories . . . 830
7.4.2 Data modeling with RDF(S) . . . 831
7.4.2.1 What is RDF? . . . 831
7.4.2.2 What is RDFS? . . . 832
7.4.3 The SPARQL query language . . . 832
7.4.3.1 What is SPARQL? . . . 832
7.4.3.2 Using SPARQL in GraphDB . . . 834
7.4.4 Create a custom graph view over your RDF data . . . 835
7.4.4.1 How does it work? . . . 835
7.4.4.2 World airport, airline, and route data . . . 835
7.4.4.3 Springer Nature SciGraph . . . 838
7.4.5 RDF-star and SPARQL-star . . . 842
7.4.5.1 The modeling challenge . . . 842
7.4.5.2 How do the different approaches compare? . . . 843
7.4.5.3 Syntax and examples . . . 844
7.4.5.4 Convert standard reification to RDF-star . . . 847
7.4.5.5 MIME types and file extensions for RDF-star in RDF4J . . . 848
7.4.6 Ontologies . . . 849
7.4.6.1 What is an ontology? . . . 849
7.4.6.2 What are the benefits of developing and using an ontology? . . . 849
7.4.6.3 Using ontologies in GraphDB . . . 850
7.4.7 Ontology mapping with the `owl:sameAs` property . . . 850
7.5 Customization and other advanced development . . . 852
7.5.1 Programming with GraphDB . . . 852
7.5.1.1 Installing Maven dependencies . . . 852
7.5.1.2 Examples . . . 852
7.5.2 Automated update notifications . . . 858
7.5.2.1 What are GraphDB local notifications? . . . 858
7.5.2.2 How to register for local notifications . . . 858
7.5.3 Extending GraphDB Workbench . . . 859
7.5.3.1 Clone, download, and run GraphDB Workbench . . . 859
7.5.3.2 Add your own page and controller . . . 859
7.5.3.3 Add repository checks . . . 860
7.5.3.4 Repository setup . . . 861
7.5.3.5 Select departure and destination airport . . . 861
7.5.3.6 Find the paths between the selected airports . . . 863
7.5.3.7 Visualize results . . . 869
7.5.3.8 Add status message . . . 869
7.5.4 Tuning query behavior . . . 870
7.5.4.1 What are named graphs . . . 870
7.5.4.2 How to manage explicit and implicit statements . . . 871
7.5.4.3 How to query explicit and implicit statements . . . 872
7.5.4.4 How to specify the dataset programmatically . . . 872
7.5.4.5 How to access internal identifiers for entities . . . 873
7.5.4.6 How to use RDF4J ‘direct hierarchy’ vocabulary . . . 874
7.5.4.7 Other query behavior customizations . . . 875
7.5.5 How the Repository configuration template works . . . 876
7.5.5.1 GraphDB repository . . . 876
7.6 Talk to Your Graph examples . . . 877
7.6.1 Star Wars example . . . 877

7.6.1.1 Creating a repository and importing the data . . . 877
7.6.1.2 Talking to your graph . . . 877
7.6.2 Acme employees and products example . . . 879
7.6.2.1 Creating a repository and importing the data . . . 879
7.6.2.2 Talking to your graph . . . 879
7.7 Vector search in nested fields with Elasticsearch . . . 883
7.7.1 Prerequisites for vector search in nested fields . . . 884
7.7.2 Creating the Elasticsearch connector . . . 884
7.7.3 Search query for similar label . . . 886
7.7.4 Search query for similar description . . . 887

8 Reference **889**
8.1 Architecture and components . . . 889
8.1.1 Architecture . . . 889
8.1.1.1 RDF4J . . . 890
8.1.1.2 The Sail API . . . 890
8.1.2 Components . . . 891
8.1.2.1 Engine . . . 891
8.1.2.2 Connectors . . . 891
8.1.2.3 Workbench . . . 892
8.2 GraphDB command line tools . . . 892
8.2.1 console . . . 892
8.2.2 generate-report . . . 892
8.2.3 graphdb . . . 893
8.2.4 importrdf . . . 893
8.2.4.1 Load command line options . . . 893
8.2.4.2 Preload command line options . . . 894
8.2.5 rdfvalidator . . . 894
8.2.6 reification-convert . . . 894
8.2.7 rule-compiler . . . 895
8.2.8 storage-tool . . . 895
8.2.8.1 Supported commands . . . 895
8.2.8.2 Options . . . 896
8.2.8.3 Examples . . . 896
8.3 Directories and configuration properties . . . 898
8.3.1 Directories . . . 898
8.3.1.1 GraphDB home . . . 898
8.3.1.2 Distribution directory . . . 899
8.3.1.3 Checking the configured directories . . . 900
8.3.2 Configuration . . . 900
8.3.2.1 Storing property values in a separate file . . . 900
8.3.2.2 Configuring GraphDB through environment variables . . . 900
8.3.2.3 Configuration properties . . . 902
8.3.2.4 Talk to Your Graph and LLM properties . . . 914
8.3.2.5 Configuring logging . . . 920
8.4 RDF formats . . . 920
8.4.1 Turtle . . . 921
8.4.2 Turtle-star . . . 921
8.4.3 TriG . . . 921
8.4.4 TriG-star . . . 921
8.4.5 N3 . . . 922
8.4.6 N-Triples . . . 922
8.4.7 N-Quads . . . 922
8.4.8 JSON-LD . . . 922
8.4.9 NDJSON-LD . . . 923
8.4.10 RDF/JSON . . . 923
8.4.11 RDF/XML . . . 923
8.4.12 TriX . . . 923

8.4.13 BinaryRDF . . . 924
8.5 SPARQL compliance . . . 924
8.5.1 SPARQL 1.1 Protocol for RDF . . . 924
8.5.2 SPARQL 1.1 Query . . . 924
8.5.3 SPARQL 1.1 Update . . . 924
8.5.3.1 Modification operations on RDF triples . . . 924
8.5.3.2 Operations for managing graphs . . . 925
8.5.4 SPARQL 1.1 Federation . . . 925
8.5.4.1 Internal SPARQL Federation . . . 925
8.5.4.2 Federated query to a remote password-protected repository . . . 925
8.5.5 SPARQL 1.1 Graph Store HTTP Protocol . . . 925
8.5.5.1 URL patterns . . . 925
8.5.5.2 Methods supported by these resources and their effects . . . 926
8.5.5.3 Request headers . . . 926
8.5.5.4 Supported parameters for requests on indirectly referenced named graphs . . . 926
8.5.6 SPARQL 1.1 non-conformance . . . 926
8.5.6.1 Deviations from SPARQL 1.1 tests . . . 928
8.6 SPARQL functions reference . . . 928
8.6.1 SPARQL 1.1 functions . . . 929
8.6.2 SPARQL 1.1 constructor functions . . . 933
8.7 SPARQL extensions reference . . . 934
8.7.1 Mathematical function extensions . . . 934
8.7.2 Date and time function extensions . . . 938
8.7.3 RDF-star extension functions . . . 940
8.7.4 SPARQL functions vs magic predicates . . . 940
8.7.5 SPARQL SPIN functions . . . 941
8.7.6 RDF list function extensions . . . 943
8.7.7 Aggregation function extensions . . . 944
8.7.8 GeoSPARQL functions . . . 944
8.7.8.1 GeoSPARQL extension functions . . . 947
8.7.9 Geospatial extension functions . . . 948
8.7.10 Jena SPARQL extensions . . . 949
8.7.11 Time functions extensions . . . 949
8.7.11.1 Period extraction functions . . . 949
8.7.11.2 Period transformation functions . . . 950
8.7.11.3 Durations expressed in certain units . . . 951
8.7.11.4 Arithmetic operations . . . 951
8.7.12 LLM GPT extensions . . . 951
8.7.13 List manipulation extension functions . . . 953
8.8 GraphDB APIs . . . 953
8.8.1 Using the RDF4J REST API . . . 954
8.8.1.1 What is the RDF4J REST API? . . . 954
8.8.1.2 Using the RDF4J REST API's SPARQL endpoints . . . 954
8.8.1.3 Graph Store HTTP Protocol . . . 956
8.8.2 Using the GraphDB REST API . . . 956
8.8.2.1 Access control list management . . . 956
8.8.2.2 Authentication . . . 957
8.8.2.3 Cluster group management . . . 957
8.8.2.4 Custom roles management . . . 957
8.8.2.5 Data import . . . 957
8.8.2.6 Location management . . . 957
8.8.2.7 Monitoring . . . 957
8.8.2.8 Recovery management . . . 958
8.8.2.9 Repository management . . . 958
8.8.2.10 SPARQL templates . . . 958
8.8.2.11 SQL views management . . . 958
8.8.2.12 Saved queries . . . 958
8.8.2.13 Security management . . . 959

8.8.3 Using GraphDB with the RDF4J API . . . 959
8.8.3.1 RDF4J API . . . 959
8.8.4 GraphDB Plugin API . . . 961
8.8.4.1 A Java GraphDB plugin . . . 961
8.8.4.2 The life cycle of a plugin . . . 963
8.8.4.3 Repository internals . . . 965
8.8.4.4 Query processing . . . 972
8.8.4.5 Update processing . . . 977
8.8.4.6 Transactions . . . 980
8.8.4.7 Exceptions . . . 981
8.8.4.8 Accessing other plugins . . . 982
8.8.4.9 List of plugin interfaces and classes . . . 983
8.8.4.10 Adding external plugins to GraphDB . . . 987
8.8.4.11 Putting it all together: example plugins . . . 988
8.8.5 Using Maven artifacts . . . 993
8.8.5.1 Public Maven repository . . . 993
8.8.5.2 Available artifacts . . . 993
8.8.5.3 RDF4J's version . . . 995
8.9 OWL compliance . . . 995
8.10 W3C specifications . . . 996
8.10.1 Resource Description Framework (RDF) . . . 996
8.10.2 SPARQL Protocol and Query Language . . . 996
8.10.3 Schemas, ontologies, and data validation . . . 997
8.11 Benchmarks . . . 997
8.11.1 LDBC Semantic Publishing Benchmark 2.0 . . . 997
8.11.1.1 Data loading . . . 998
8.11.1.2 Production load . . . 999
8.11.2 Berlin SPARQL Benchmark (BSBM) . . . 1000
8.12 GraphDB system statements . . . 1000
8.12.1 System graphs . . . 1000
8.12.2 System predicates . . . 1002
8.13 Glossary . . . 1002
8.14 JavaScript functions . . . 1004
8.14.1 Registering JavaScript functions . . . 1004
8.14.2 Removing JavaScript functions . . . 1005

9 Help **1007**
9.1 FAQ . . . 1007
9.1.1 General . . . 1007
9.1.1.1 What is OWLIM? . . . 1007
9.1.1.2 Why a solid-state drive and not a hard-disk one? . . . 1007
9.1.1.3 Is GraphDB Jena-compatible? . . . 1007
9.1.1.4 Can GraphDB run on Redhat or Centos? . . . 1007
9.1.1.5 Can GraphDB load archived files? . . . 1007
9.1.1.6 Can I have a GraphDB Free Docker image? . . . 1008
9.1.1.7 Is there a Python/Java/C#/Javascript/REST API? . . . 1008
9.1.2 Configuration . . . 1008
9.1.2.1 How do I find out the exact version number of GraphDB? . . . 1008
9.1.2.2 What is a location? . . . 1008
9.1.2.3 How do I attach a location? . . . 1008
9.1.2.4 Does GraphDB do backups on its own? . . . 1009
9.1.3 RDF and SPARQL . . . 1009
9.1.3.1 How is GraphDB related to RDF4J? . . . 1009
9.1.3.2 What does it mean when an IRI starts with `urn:rdf4j:triple:`? . . . 1009
9.1.3.3 What kind of SPARQL compliance is supported? . . . 1009
9.1.3.4 Can I use SHACL with a specific context only? . . . 1009
9.1.3.5 Can I have partial inserts with SHACL? . . . 1010
9.1.3.6 Can I get performance information about my queries? . . . 1010

9.1.3.7 Can I create groups of saved queries? . . . 1010
9.1.3.8 Can I export a single named graph? . . . 1010
9.1.4 Security . . . 1010
9.1.4.1 Does GraphDB have any security vulnerabilities? . . . 1010
9.1.4.2 Does the Log4Shell issue (CVE-2021-44228) affect GraphDB? . . . 1010
9.1.5 Troubleshooting . . . 1010
9.1.5.1 Why can't I use custom rule file (`.pie`) — an exception occurred? . . . 1010
9.1.5.2 Why can't I open GraphDB in MacOS? . . . 1010
9.1.5.3 How to workaround "Insufficient disk space to start a transaction for repository" error? . . . 1011
9.1.5.4 The login form does not show up, security is on, and I cannot log in. What to do? 1011
9.1.5.5 Why is the database storage not getting smaller when I delete triples? . . . 1011
9.2 Support . . . 1011
9.3 Release notes . . . 1011
9.3.1 GraphDB 11.3.3 . . . 1012
9.3.1.1 Component versions . . . 1012
9.3.1.2 Core database . . . 1012
9.3.1.3 Cluster . . . 1012
9.3.1.4 Security . . . 1012
9.3.2 GraphDB 11.3.2 . . . 1013
9.3.2.1 Component versions . . . 1013
9.3.2.2 Core database . . . 1013
9.3.2.3 Cluster . . . 1014
9.3.2.4 Security . . . 1014
9.3.2.5 GraphQL . . . 1015
9.3.2.6 Workbench & UX . . . 1015
9.3.3 GraphDB 11.3.1 . . . 1015
9.3.3.1 Component versions . . . 1016
9.3.3.2 Core database . . . 1016
9.3.3.3 Cluster . . . 1016
9.3.3.4 Security . . . 1017
9.3.3.5 GraphQL . . . 1017
9.3.3.6 LLM & AI . . . 1017
9.3.3.7 Workbench & UX . . . 1018
9.3.4 GraphDB 11.3.0 . . . 1019
9.3.4.1 Component versions . . . 1019
9.3.4.2 Core database . . . 1020
9.3.4.3 Security . . . 1020
9.3.4.4 GraphQL . . . 1021
9.3.4.5 LLM & AI . . . 1021
9.3.4.6 External integrations . . . 1021
9.3.4.7 Workbench & UX . . . 1022
9.4 Third-party vulnerabilities . . . 1022

CHAPTER

ONE

GETTING STARTED WITH GRAPHDB

1.1 What is GraphDB?

Good places to start for…

- role-everyone: All users, especially new ones, who can benefit from introductions to basic GraphDB capabilities.
- role-knowledge-engineer: People who focus on the technologies and standards for creating and taking advantage of knowledge graphs.
- role-data-scientist: People who use a variety of data formats and tools, often in different combinations, to identify patterns that help to answer questions.
- role-developer: People who are building an application that uses GraphDB as a backend.
- role-admin: People who are installing and maintaining a GraphDB system or an application that uses GraphDB as the backend.

Ontotext GraphDB is a highly efficient, scalable and robust *graph database* with *RDF* and *SPARQL* support. GraphDB is one of the few triplestores that can perform real-time *semantic inferencing* at scale, allowing users to derive new semantic facts from existing facts.

With excellent enterprise features, integration with external search applications, LLM Graph Retrieval-Augmented Generation (Graph RAG) capabilities, and compatibility with industry standards, GraphDB is the preferred database choice of both small independent developers and big enterprises.

Highlights of GraphDB include:

- Dedicated community and commercial support.
- Integration with Elasticsearch, OpenSearch, Lucene, Solr, and other high-performance search applications.
- Clusters and multi-region deployments support.
- *Connectors*, which provide fast keyword and faceted searches that are typically implemented by an external component. All GraphDB editions include connectors for Lucene and ChatGPT, and the Enterprise Edition includes connectors for Solr, Elasticsearch, OpenSearch, and Kafka.
- Implementation of the RDF4J framework interfaces and the W3C SPARQL Protocol specification. Learn more about RDF4J and the roles of its layers in the *Architecture and components* chapter of this documentation's Reference section.
- Support for all RDF serialization formats.
- Availability as a *cloud deployment* on *Amazon Web Services* and on *Microsoft Azure*.

Reasoning and query evaluation are performed over a persistent storage layer. Loading, *reasoning*, and query evaluation proceed extremely quickly even against huge *ontologies* and *knowledge bases*.

GraphDB can manage billions of explicit statements on desktop hardware and handle tens of billions of statements on commodity server hardware. According to the LDBC Semantic Publishing Benchmark, it is one of the most scalable OWL repositories currently available.

Ontotext offers licenses for two editions of GraphDB:

- GraphDB Free
- GraphDB Enterprise (EE)

The GraphDB Standard Edition (SE), while no longer offered, still receives legacy support.

1.1.1 Licensing

GraphDB uses RDF4J as a library, utilizing its APIs for storage and querying, as well as the support for a variety of query languages such as SPARQL and SeRQL and RDF syntaxes such as Turtle and RDF/XML. Full licensing information is available in the license files located in the doc folder of the distribution package.

GraphDB is offered under several different licenses, including a free-to-use non-commercial option.

- **GraphDB Free** is a free-to-use semantic repository system, suitable for massive volumes of data. It employs file-based indexes that make it possible to scale to billions of statements even on desktop machines, and ensures fast query evaluations through *inference* and *query optimizations*. GraphDB Free supports a limit of two concurrent queries, and is suitable for low query loads and smaller projects.

  > **Note:** Starting from GraphDB 11.0, GraphDB Free is no longer automatically distributed with GraphDB. If you want to use GraphDB Free, you must first request a free license from the Graphwise website and manually install it before use.
  >
  > This also applies when upgrading between minor versions of GraphDB.

- The **GraphDB Enterprise Edition (EE)** is a high-performance, clustered *semantic repository* that scales in production environments with simultaneous loading, querying and inferencing of billions of RDF statements. It supports a *high-availability cluster* based on the Raft consensus algorithm, with several features that are crucial for achieving enterprise-grade highly available deployments. It also adds more connectors for full-text search and faceting — the *Solr* and *Elasticsearch* connectors, as well as the *Kafka connector* for synchronizing changes to the RDF model to any Kafka consumer.

  The Enterprise Edition license is offered on a per-server-CPU-core basis. In other words each license is applicable to an individual machine only, and for a specific number of CPU cores on that machine. For example, when running a *cluster*, each individual node must be licensed separately.

  The number of cores affects the performance of your server — the more cores your license supports, the more your GraphDB deployment will be optimized. Excluding internal federation, each individual core supports only a single concurrent query. In other words, the number of concurrent queries you can run is equal to the number of cores you are licensed for. The number of cores also determines how backup and restore use parallel streams with cloud storage and it affects the efficiency of importing data and the preload tool.

GraphDB Standard Edition (SE) is a discontinued commercial version of GraphDB with a smaller feature set than GraphDB Enterprise. It is no longer offered but is still supported.

1.1.1.1 GraphDB feature comparison

Feature	GraphDB Free	GraphDB Enterprise
Deployment type	On-premise only	On-premise, Managed service
GraphDB core features	✓	✓
GraphQL basic (core feature)	✓	✓
GraphQL extended (add-on for an additional cost)	×	×
GPT SPARQL Functions	✓	✓
Talk to Your Graph	✓	✓
Concurrency	single core	multiple cores
Advanced security: LDAP	×	✓
Advanced security: Kerberos	×	✓
Advanced security: X.509 certificate	×	✓
Advanced security: OpenID/OAuth	×	✓
Advanced security: FGAC	×	✓
Additional connectors: Elasticsearch	×	✓
Additional connectors: OpenSearch	×	✓
Additional connectors: Solr	×	✓
Additional connectors: Kafka	×	✓
Support	Community	SLA
High-availability cluster	×	✓
40-bit entity IDs (more than 2B RDF nodes)	×	✓
Max number of repositories	5	Unlimited

GraphDB core features

All editions of GraphDB support the following core features:

- Full *SPARQL 1.1 support*
- Management of an unlimited number of RDF statements
- 100% compatibility with RDF4J framework
- Ultra fast *forward-chaining reasoning*
- Efficient *retraction of inferred statements* upon update
- Full standard-compliant and optimized *rulesets* for RDFS, OWL 2 RL, and QL
- Custom *reasoning* and *consistency checking rulesets*
- *Plugin API* for engine extension
- Support for *Geospatial indexing* and querying, plus *GeoSPARQL*
- *Query optimizer* allowing effective query execution
- *Workbench interface* to manage repositories, data, user accounts and access roles
- *Lucene* connector for full-text search

GraphQL extended features

These extended GraphQL features are not included in any GraphDB edition and are available as a separate add-on for an additional cost:

- Role Based Access Control (RBAC)
- Schema validation against data
- GraphQL federation
- SPARQL Federation

1.1.2 Cloud deployments for GraphDB

GraphDB is available as a cloud deployment on the following platforms:

- **Amazon Web Services**: See *How to deploy GraphDB in AWS* and *general installation instructions* for information on using GraphDB with your AWS account.
- **Microsoft Azure**: See *How to deploy GraphDB in Azure* and *general installation instructions* for information on using GraphDB with your Azure account.

1.2 How to install GraphDB

GraphDB can be operated as a desktop or a server application. The server application is recommended if you plan to migrate your setup to a production environment. Choose the one that best suits your needs, and follow the steps below:

- *Run GraphDB as a desktop installation* — For desktop users, we recommend the quick installation, which comes with a preconfigured Java runtime environment. This is the easiest and fastest way to start using the GraphDB database.
- *Run GraphDB as a standalone server* — For production use, we recommend installing the standalone server. The installation comes with a preconfigured web server. This is the standard way to use GraphDB if you plan to use the database for longer periods with preconfigured log files.
- *Run GraphDB in the AWS cloud* — GraphDB is also available on Amazon Web Services. This installation requires you to set up an environment on AWS before you install GraphDB.
- *Run GraphDB in the Azure cloud* — GraphDB is also available on Microsoft Azure. This installation requires you to set up an environment on Azure before you install GraphDB.

GraphDB is available under two different licenses: Free and Enterprise. Regardless of the edition, see *Set up GraphDB license*.

To see the content of the GraphDB distribution package, see the *Distribution package documentation*.

Note: Check the *Migrating GraphDB configurations* guide for help with updating your GraphDB version.

1.2.1 Requirements

Note: In addition to the technical requirements outlined below, some GraphDB features also require an **Enterprise Edition** license. Check the *Licensing* documentation for further information on the different types of GraphDB licenses.

1.2.1.1 Minimum requirements

The minimum requirements allow loading datasets of only up to 50 million RDF triples.

- 3GB of memory
- 8GB of storage space
- Java SE Development Kit 21 (not required for GraphDB Free desktop installation)

  Note: While newer versions than Java 21 should generally work, GraphDB 11 does not officially support Java versions above 21.

Warning: All GraphDB indexes are optimized for hard disks with very low seek time. We strongly recommend using only SSD storage for persisting repository images.

We strongly advise against the use of network file system (NFS) storage. With NFS, updated files are written on the file system in full. This results in significant update times for repositories with 10M or more statements, especially when processing large transactions. For comparison, block storage file systems need to access only the updated blocks.

1.2.1.2 Hardware sizing

The best approach for correctly sizing the hardware resources is to estimate the number of explicit statements. Statistically, an average dataset has a 3:1 ratio of statements to unique RDF resources. (If you need advice about estimating the total number of resources to plan for, contact your Ontotext representative.) The total number of statements determines the expected repository image size, and the number of unique resources affects the memory footprint required to initialize the repository.

The table below summarizes the recommended parameters for planning RAM and disk sizing:

- **Statements** are the planned number of explicit + implicit statements.
- **Java heap (minimal)** is the minimal recommend JVM heap required to operate the database controlled by `-Xmx` parameter.
- **Java heap (optimal)** is the recommended JVM heap required to operate a database controlled by `-Xmx` parameter.
- **OS** is the recommended minimal RAM reserved for the operating system.
- **Total** is the RAM required for the hardware configuration.
- **Repository image** is the expected size on disk of your repositories. For repositories with inference, use the total number of explicit + implicit statements.

Note: Implicit statements are regular statements that are materialized and their number (expansion ratio) can vary depending on the specific data and chosen ruleset.

Statements	Java heap (min)	Java heap (opt)	Off-heap	OS	Total	Repository image
150M	5GB	6GB	1GB	2GB	9GB	17GB
300M	8GB	12GB	2GB	3GB	17GB	34GB
750M	12GB	16GB	3GB	4GB	23GB	72GB
1.5B	32GB	32GB	7GB	4GB	43GB	150GB
3B	50GB	58GB	12GB	4GB	74GB	350GB
7.5B	64GB	68GB	14GB	4GB	86GB	720GB
15B	80GB	88GB	17GB	4GB	109GB	1450GB
30B	128GB	128GB	25GB	6GB	159GB	2900GB

Note: If you are planning on using the *external cluster proxy* feature, we recommend you have at least 1GB RAM and a single core. In Kubernetes, 500m for CPU would suffice.

Warning: Running a repository in a cluster doubles the requirements for the repository image storage. The table above provides example sizes for a single repository and does not take restoring backups or snapshot replication in consideration.

1.2.1.3 Memory management

The optimal approach towards memory management of GraphDB is based on a balance of performance and resource availability per repository. In heavy use cases such as parallel importing into a number of repositories, GraphDB may take up more memory than usual.

There are several configuration properties with which the amount of memory used by GraphDB can be controlled:

- Reduce the global cache: by default, it can take up to 40% (or up to 40GB in case of heap sizes above 100GB) of the available memory allocated to GraphDB, which during periods of stress can be critical. By reducing the size of the cache, more memory can be freed up for the actual operations. This can be beneficial during periods of prolonged imports as that data is not likely to be queried right away.

  ```
  graphdb.page.cache.size=2g
  ```

- Reduce the buffer size: this property is used to control the amount of statements that can be stored in buffers by GraphDB. By default, it is sized at 200,000 statements, which can impact memory usage if many repositories are actively reading and writing data at once. The optimal buffer size depends on the hardware used, as reducing it would cause more write and read operations to the actual storage.

  ```
  pool.buffer.size=50000
  ```

- Disable parallel import: during periods of prolonged imports to a large number of repositories, parallel imports can take up more than 800 megabytes of retained heap per repository. In such cases, parallel importing can be disabled, which would force data to be imported serially to each repository. However, serial import reduces performance.

  ```
  graphdb.engine.parallel-import=false
  ```

This table shows an example of retained heap usage by repository, using different configuration parameters:

Configurations	Retained heap per repository	
	During prolonged import	Stale
Default	≥800MB	340MB
+ Reduced global cache (2GB)	670MB	140MB
+ Reduced buffer size[1]	570-620MB	140MB
+ Reduced inference pool size*	370-550MB	140MB
Serial import[2]	210-280MB	140MB

** Without reducing buffer and inference pool sizes. Disables parallel import, which impacts performance.

1.2.2 GraphDB Desktop installation

The easiest way to set up and run GraphDB is to use the native installations provided for the GraphDB Desktop distribution. This kind of installation is the best option for your laptop or desktop computer and does not require the use of a console, as it works in a graphic user interface (GUI). For this distribution, you do not need to download Java, as it comes bundled together with GraphDB.

Go to the GraphDB download page and request your GraphDB copy. You will receive an email with the download link. According to your OS, proceed as follows:

Important: **GraphDB Desktop** is a new application that is similar to but different from the previous application **GraphDB Free**.

If you are upgrading from the old GraphDB Free application, you need to stop GraphDB Free and uninstall it before or after installing GraphDB Desktop. Once you run GraphDB Desktop for the first time, it will convert some of the data files and GraphDB Free will no longer work correctly.

1.2.2.1 On Windows

1. Download the GraphDB Desktop `.msi` installer file.
2. Double-click the application file and follow the on-screen installer prompts.
3. Locate the GraphDB Desktop application in the Windows Start menu and start it. The GraphDB Workbench opens in your browser at http://localhost:7200/.

1.2.2.2 On MacOS

1. Download the GraphDB Desktop `.dmg` file.
2. Double-click it to create a virtual disk on your desktop. Copy the program from the virtual disk to your hard disk *Applications* folder, and you're set.
3. Start GraphDB Desktop by clicking the application icon. The GraphDB Workbench opens in your browser at http://localhost:7200/.

1.2.2.3 On Linux

1. Download the GraphDB Desktop `.deb` or `.rpm` file.
2. Install the package with `sudo dpkg -i` or `sudo rpm -i` and the name of the downloaded package. Alternatively, you can double-click the package name.
3. Start GraphDB Desktop by clicking the application icon. The GraphDB Workbench opens in your browser at http://localhost:7200/.

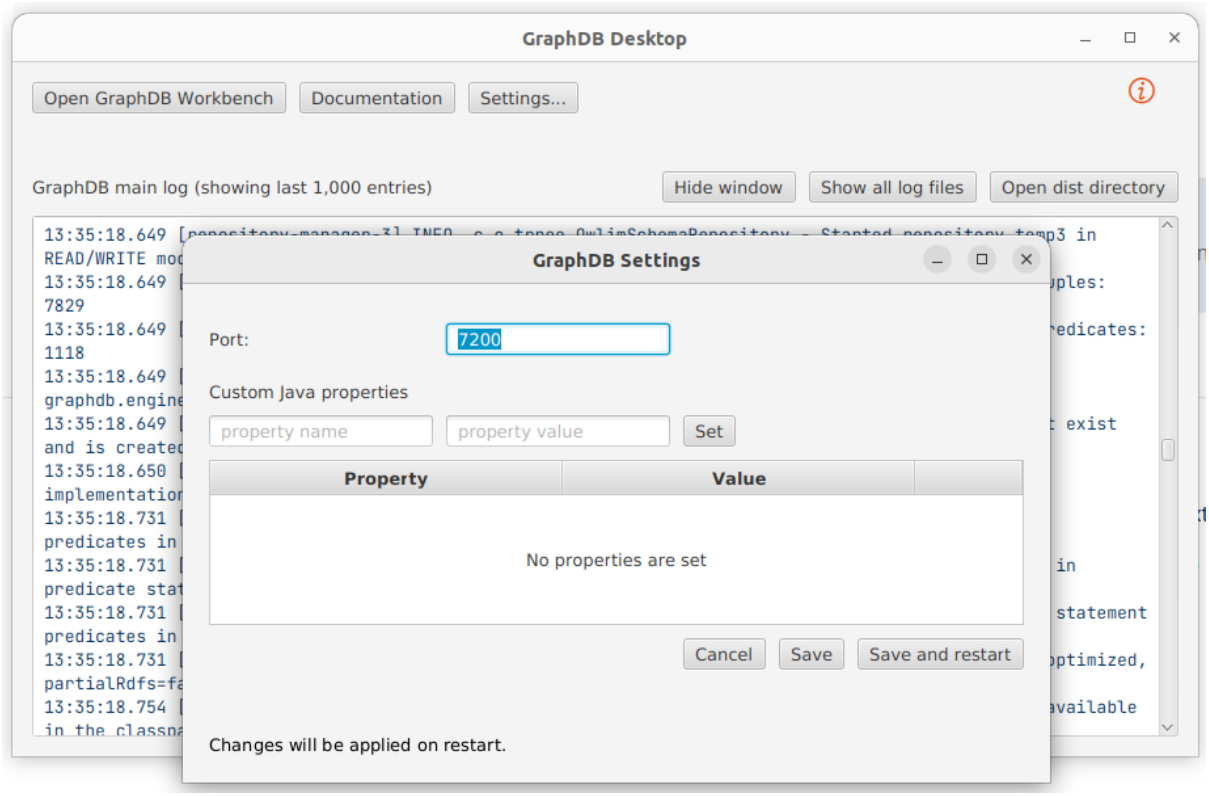
[1] Depends on the number of CPU cores that are available to GraphDB. For the statistics, the default buffer size was reduced from 200,000 (default) to 50,000 statements. The inference pool size was reduced from eight to three. Keep in mind that this reduces performance.

[2] Without reducing buffer and inference pool sizes. Disables parallel import, which impacts performance.

1.2.2.4 Configuring your running GraphDB instance

Once GraphDB Desktop is running, a small icon appears on your status bar, menu, or tray area, depending on your operating system. It lets you check whether the database is running and opens the GraphDB Desktop application window, which has buttons that let you do the following:

- *Open GraphDB Workbench*: open the Workbench in a new browser tab.
- *Documentation*: open the GraphDB documentation's home page in a new browser tab.
- *Settings…*: open a window that lets you reset the port where GraphDB is running and edit any custom Java properties, as shown below.
- *Hide window*: close the application window. You can reopen it with the icon mentioned above.
- *Show all log files*: open the `.graphdb/logs` folder.
- *Open dist directory*: open the *Distribution directory* folder.



Closing the application window will, after confirming that you want to do so, shut down GraphDB.

Configuring the JVM

You can add and edit the JVM options (such as Java system properties or parameters to set memory usage) of the GraphDB native app from the GraphDB Desktop config file. It is located at:

- On Mac: `/Applications/GraphDB Desktop.app/Contents/app/GraphDB Desktop.cfg`
- On Windows: `\Users\<username>\AppData\Local\GraphDB Desktop\app\GraphDB Desktop.cfg`
- On Linux: `/opt/graphdb-desktop/lib/app/graphdb-desktop.cfg`

The JVM options are defined at the end of the file and will look very similar to this:

```
[JavaOptions]
java-options=-Djpackage.app-version=10.0.0
java-options=-cp
```

(continues on next page)

(continued from previous page)

```
java-options=$APPDIR/graphdb-native-app.jar:$APPDIR/lib/*
java-options=-Xms1g
java-options=-Dgraphdb.dist=$APPDIR
java-options=-Dfile.encoding=UTF-8
java-options=--add-exports
java-options=jdk.management.agent/jdk.internal.agent=ALL-UNNAMED
java-options=--add-opens
java-options=java.base/java.lang=ALL-UNNAMED
```

Each `java-options=` line provides a single argument passed to the JVM when it starts. To be on the safe side, we recommend that you do not remove or change any of the existing options provided with the installation. You can add your own options at the end. For example, if you want to run GraphDB Desktop with 8 gigabytes of maximum heap memory, you can set the following option:

```
java-options=-Xmx8g
```

1.2.2.5 Stopping GraphDB

To stop the database, simply quit it from the status bar, menu, or tray area icon or close the GraphDB Desktop application window.

Tip: Some Linux systems do not support a status bar, menu, or tray area. If you have hidden the GraphDB window, you can quit it by killing the process the same way that the GraphDB *standalone server* gets killed.

1.2.3 GraphDB standalone server

The default way to run GraphDB is as a standalone server. The server is platform-independent, and includes all recommended JVM (Java Virtual Machine) parameters for immediate use.

Note: Before downloading and running GraphDB, please make sure to have JDK (Java Development Kit, recommended) or JRE (Java Runtime Environment) installed. GraphDB requires Java 21 or greater.

1.2.3.1 Running GraphDB

1. Download the GraphDB distribution file and unzip it.
2. Start GraphDB by executing the `graphdb` startup script located in the `bin` directory of the GraphDB distribution.

   A message appears in the console telling you that GraphDB has been started in Workbench mode. To access the Workbench, open http://localhost:7200/ in your browser.

See the supported startup script options in the *graphdb command line tool documentation*.

1.2.3.2 Configuring GraphDB

Paths and network settings

The configuration of all GraphDB directory paths and network settings is read from the `conf/graphdb.properties` file. It controls where to store the database data, log files, and internal data. To assign a new value, modify the file or override the setting by adding `-D<property>=<new-value>` as a parameter to the startup script. For example, to change the database port number:

```
graphdb -Dgraphdb.connector.port=<your-port>
```

The configuration properties can also be set in the environment variable `GDB_JAVA_OPTS`, using the same `-D<property>=<new-value>` syntax.

Note: The order of precedence for GraphDB configuration properties is:

1. arguments supplied on the command line
2. `GDB_JAVA_OPTS`
3. configuration file

The GraphDB home directory

The GraphDB home defines the root directory where GraphDB stores all of its data. The home can be set through the system or config file property `graphdb.home`.

The default value for the GraphDB home directory depends on how you run GraphDB:

- Running as a standalone server: the default is the same as the distribution directory.
- All other types of installations: OS-dependent directory.
  - On Mac: `~/Library/Application Support/GraphDB`.
  - On Windows: `\Users\<username>\AppData\Roaming\GraphDB`.
  - On Linux and other Unixes: `~/.graphdb`.

GraphDB does not store any files directly in the home directory, but uses several *subdirectories* for data or configuration.

Java Virtual Machine settings

We strongly recommend setting explicit values for the Java heap space. You can control the heap size by supplying an explicit value to the startup script such as `graphdb -Xms10g -Xmx10g` or by setting one of the following environment variables:

- `GDB_HEAP_SIZE`: both the minimum and the maximum heap size (recommended)
- `GDB_MIN_MEM`: only the minimum heap size
- `GDB_MAX_MEM`: only the maximum heap size

For more information on how to change the default Java settings, check the instructions in the `bin/graphdb` file.

Note: The order of precedence for JVM options is:

1. arguments supplied on the command line line
2. `GDB_JAVA_OPTS`
3. `GDB_HEAP_SIZE`

4. `GDB_MIN_MEM/GDB_MAX_MEM`.

Tip: Every JDK package contains a default garbage collector (GC) that can potentially affect performance. We benchmarked GraphDB's performance against the LDBC SPB and BSBM benchmarks with JDK 8 and 11. With JDK 8, the recommended GC is Parallel Garbage Collector (ParallelGC). With JDK 11, you'll get the best performance with either G1 GC or ParallelGC.

1.2.3.3 Stopping the database server

To stop the database, find the GraphDB process identifier and send `kill <process-id>`. This sends a shutdown signal and the database stops. If the database is run in non-daemon mode, you can also send `Ctrl+C` interrupt to stop it.

Warning: We recommend against including the `-9` argument with the `kill` command because this makes a clean shutdown less likely.

1.2.4 Structure of the distribution package

The GraphDB platform-independent distribution packaged in version 7.0.0 and newer contains the following files:

Path	Description
`adapters/`	Support for SAIL graphs with the Blueprints API
`benchmark/`	Semantic publishing benchmark scripts
`bin/`	Scripts for running various utilities such as ImportRDF and the Storage Tool
`conf/`	GraphDB properties and logback.xml
`configs/`	Standard reasoning rulesets and a repository template
`doc/`	License agreements
`examples/`	Getting started and Maven installer examples, sample dataset, and queries
`lib/`	Database binary files
`plugins/`	GeoSPARQL and SPARQL-mm plugins
`README`	The readme file Custom admin handler for the Solr Connectors
`tools/`	Custom admin handler for the Solr Connectors

The first successful database run will generate the following directories, unless their default value is explicitly changed in `conf/graphdb.properties`:

Default path	Description
`data/`	Repository data
`logs/`	Database log files
`work/`	Non-user-editable configurations

1.3 Working with Workbench

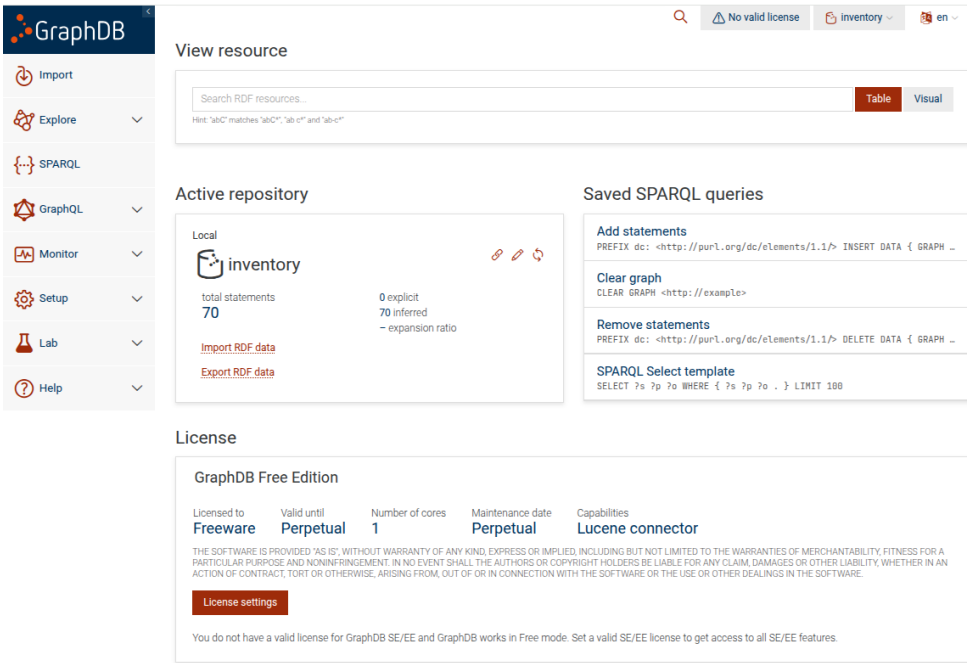
What makes GraphDB Workbench different?

- Better SPARQL editor based on YASGUI
- Import of server files
- Export in more formats
- Query monitoring with the possibility to kill a long-running query
- System resource monitoring
- User and permission management
- Connector management
- Cluster management

The Workbench is the web-based administration interface to GraphDB. It lets you administer GraphDB as well as load, transform, explore, manage, query, and export data.

The Workbench layout consists of two main areas. The navigation area is on the left-hand side of the screen and contains drop-down menus that lead to all tools and features: *Import*, *Explore*, *SPARQL*, *Monitor*, *Setup*, *Lab*, and *Help*. The work area lets you interact with the tool selected on the left. The home page provides easy access to some of the actions in the Workbench such as selecting or creating a repository. If a repository is already selected, the home page also shows information about that repository, saved SPARQL queries, and a *View resource* field to explore resources in that repository. At the bottom of the page, you can see the license details, and in the footer, the versions of the various GraphDB components.

When working with Workbench, you can also share links to the current view. When such a link is loaded, you will get a prompt whether you want to automatically change the active repository in Workbench to the one you were working on.



The Workbench officially supports the following browsers:

Browser	Support
Chrome	Full
Microsoft Edge	Full
Safari	Full
Mozilla Firefox	Best effort support

The GraphDB Workbench does not officially support other browsers or mobile browsers.

Navigation Tab	Description
Import	Import data from local files, from files on the server where the Workbench is located, from a remote URL (with a format extension or by specifying the data format), or by pasting the RDF data in the *Text area* tab. Each import method supports different serialization formats.
Explore	• *Graphs overview* — See a list of the default graph and all named graphs in the currently connected repository. Use it to inspect the statements in each graph, export them or clear the graph's data. • *Class hierarchy* — Explore the hierarchy of RDF classes by number of instances. The biggest circles are the parent classes and the nested ones are their children. Hover over a given class to see its subclasses or zoom in a nested circle (RDF class) for further exploration. • *Class relationships* — Explore the relationships between RDF classes, where a relationship is represented by links between the individual instances of two classes. Each link is an RDF statement where the subject is an instance of one class, the object is an instance of another class, and the link is the predicate. Depending on the number of links between the instances of two classes, the bundle can be thicker or thinner; it gets assigned the color of the class with more incoming links. The links can be in both directions. • *Visual graph* — Explore your data graph visually. Start from a single resource and the resources connected to it or from a graph query result. Click on a resource to expand its connections. • *Similarity* — Look up semantically similar entities and text.
SPARQL	• *SPARQL* — Query and update your data. Use any type of SPARQL query and click Run to execute it.
GraphQL	• *Endpoint Management* — Create and manage endpoints that provide GraphQL interfaces to your GraphDB repositories. • *GraphQL Playground* — Enter and edit GraphQL queries for your GraphQL endpoints on an interactive form, execute those queries, and see the results.

Continued on next page

Table 1 – continued from previous page

Navigation Tab	Description
Monitor	• *Queries and Updates* — Monitor all running queries or updates in GraphDB. Any query or update can be killed by pressing the *Abort* button displayed with the information about the running query. • *Backup and Restore* — Monitor all running backups and restores in GraphDB. • *System* — Monitor: – The usage of various system resources such as system CPU load, file descriptors, heap memory usage, off-heap memory usage, and disk storage. – The performance of queries, global page cache, entity pool, and transactions and connections. – Cluster health, if a cluster is running.
Setup	• *Repositories* — Manage repositories and connect to remote locations. A location represents a local or remote instance of GraphDB. Only a single location can be active at a given time. • *Users and Access* — Manage users and their access to the GraphDB repositories. You can also enable or disable the security of the entire Workbench. When disabled, everyone has full access to the repositories and the admin capabilities. • *My Settings* — Configure the default behavior of the Workbench. • *Connectors* — Create and manage GraphDB Connector instances. • *Cluster* — Manage a GraphDB cluster — create or modify a cluster by dragging and dropping the nodes, or use it to monitor the state of a running cluster in near real time. The view shows repositories from the active location and all remote locations. **Note:** This feature requires a *GraphDB Enterprise license*. • *Plugins* — activate or deactivate the plugins that add optional features. • *Namespaces* — View and manipulate the RDF namespaces for the active repository. You need a write permission to add or delete namespaces. • *Autocomplete* — Enable or disable the autocomplete index and check its status. This is used for automatic completion of URIs in the SPARQL editor and the View Resource page. • *RDF Rank* — Identify the more important or popular entities in your repository by examining their interconnectedness determined by the RDF Rank algorithm. Their popularity can then be used to order query results. • *JDBC* — Configure the JDBC driver to allow SQL access to repository data. • *SPARQL Templates* — Create and store predefined SPARQL templates for future updates of repository data. • *License* — View the details of your current GraphDB license and set or revert to a different one.
Lab	• *Talk to Your Graph* — Form-based interface to submit natural language queries to GraphDB's Talk to Your Graph feature.

Continued on next page

Table 1 – continued from previous page

Navigation Tab	Description
Help	• *Interactive guides* — A set of tutorials that lead you through various GraphDB features using the Workbench user interface. • *REST API* — REST API documentation of all available public RESTful endpoints together with an interactive interface for executing requests. • *Documentation* — Link to the GraphDB public documentation. • *Tutorials* — A list of tutorials that show you how to load and query data, explore data and class relationships, and visualize data. • *Support* — Link to the GraphDB support page. • *System information* — Information about the release of GraphDB, your operating system, and various components in use: – Application Info – JVM Arguments – Workbench Configuration properties **Note:** From this screen, you can also generate a detailed server report file that you can use to hunt down issues.

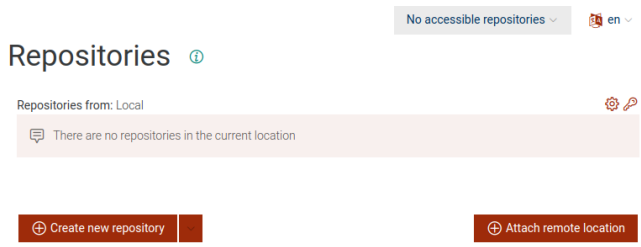
GraphDB Workbench is a separate project available at https://github.com/Ontotext-AD/graphdb-workbench. It is also part of the GraphDB distribution and can be configured with the `graphdb.workbench.home` property. As a user, this makes it easy for you to extend and reuse parts of the Workbench. See *Extending GraphDB Workbench* for more on this.

1.3.1 Create your first repository using Workbench

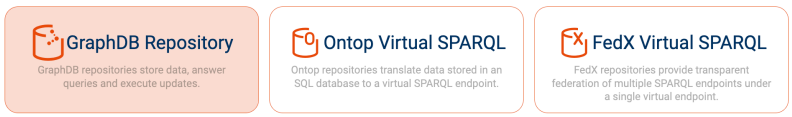
GraphDB's Workbench, console, and APIs offer a choice of ways to create and manage repositories. For your first one, it's simplest to use Workbench's web-based graphical user interface.

To manage your repositories, go to *Setup* ▸ *Repositories*. This opens a list of available repositories and their locations.

1. Click the *Create new repository* button:



2. Select *GraphDB repository*:



3. Enter the *Repository ID* (e.g., my_repo) and leave all other optional configuration settings with their default values.
4. Click the *Create* button. Your new repository will appear in the repository list.

Note: GraphDB offers a wide range of options in where and how you create your repositories.

- If your GraphDB instance is connected to another one over a network, you can also *create a repository on that remote location*.
- To learn how to create a repository with the RDF4J `console` utility, see the *console* section of the "GraphDB command line tools" documentation.
- For repositories with over several tens of millions of statements, see *the configuration parameters*.

1.3.2 Load and query some data

1.3.2.1 Query data through the Workbench

SPARQL is the W3C standard query language for RDF graph databases. (If you are unfamiliar with SPARQL you can learn more at the Graphwise Academy.) Below, after creating a repository and loading some data into it, we're going to issue a SPARQL query against that data and see the results.

First, create a GraphDB repository as described in the previous section, *Create your first repository using Workbench*. For this new repository, enter "starwars" in its *Repository ID* field and "Star Wars data" in its *Repository description* field.

After you have created it and see *starwars* listed on the *Repositories* screen, it won't necessarily be the currently connected one. The *Connected repository* icon shows which is currently connected and the *Connect repository* icon shows which ones are available but not connected. If necessary, click the *Connect repository* icon next to *starwars*.

To load a sample dataset into this repository:

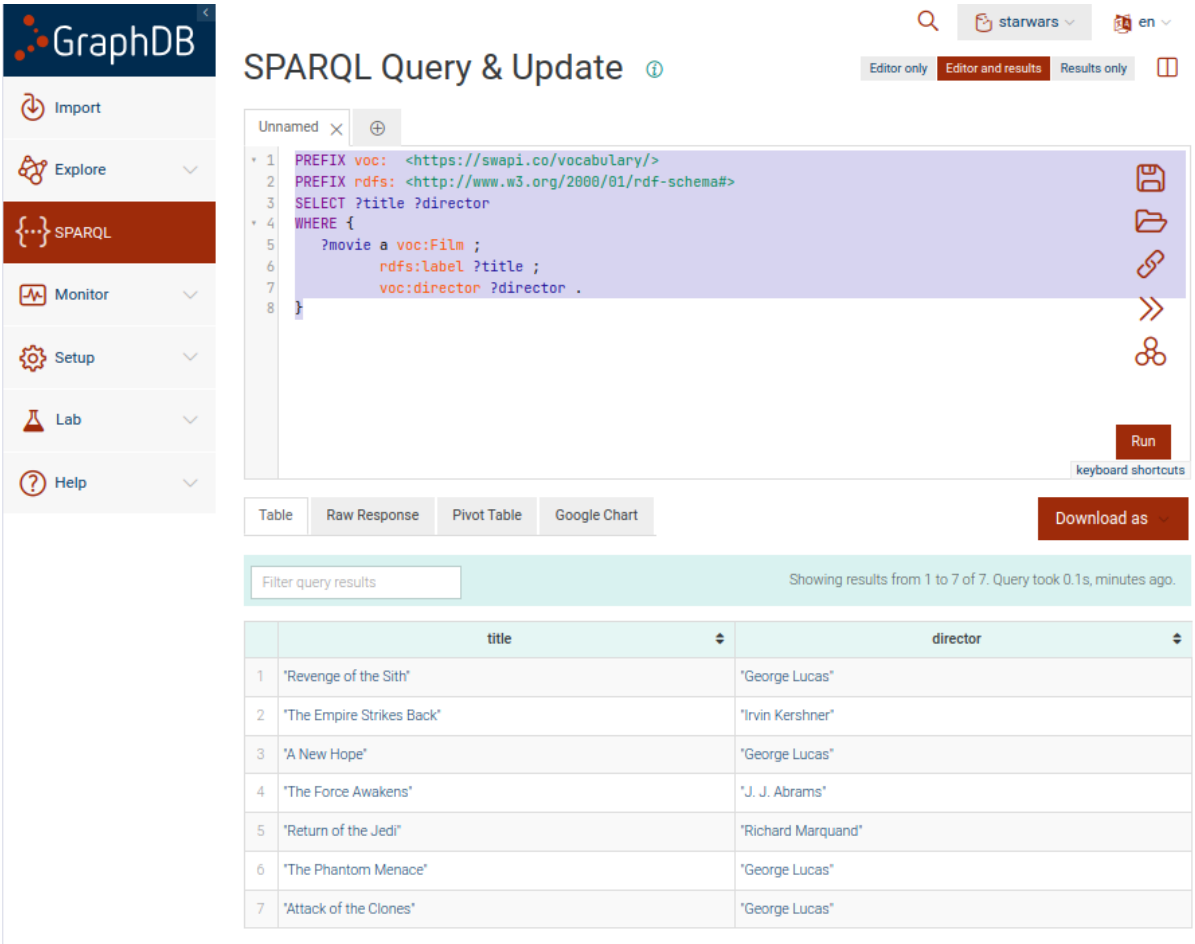
1. Download the `starwars.ttl` file and remember where you stored it.
2. Select *Import* from the Workbench's main menu on the left.
3. On the *Import* screen, select *Upload RDF files*, navigate to the folder where you stored the downloaded file, and select it. This will add the file to the *Import* screen with a checkbox on its left.
4. Click the *Import* button on the right of the *starwars.ttl* entry on the screen's list.
5. Click *Import* on the *Import settings* dialog box. This should return you to the *Import* screen, where the message "Imported successfully in less than a second" gets added to the *starwars.ttl* row.

Let's query the data that we've loaded into this repository.

1. Click *SPARQL* on the Workbench's main menu.
2. Paste the following SPARQL query into the query field. It looks for instances of the `voc:Film class`, and for each one it finds, it stores the `rdfs:label` value in the `?title` variable and its `voc:director` value in the `?director` variable:

```
PREFIX voc:  <https://swapi.co/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?title ?director
WHERE {
   ?movie a voc:Film ;
          rdfs:label ?title ;
          voc:director ?director .
}
```

3. Click the *Run* button in the lower right of the field. You will see the list of titles and directors that your query requested in the result area underneath the query:



Congratulations! Your copy of GraphDB is up and running and waiting for more queries, data, and enrichment of that data as you build knowledge graphs to drive your applications. See the *Tutorials* section in this documentation's main menu to learn more.

1.3.2.2 Query data programmatically

SPARQL is not only a standard query language, but also a protocol for communicating with RDF databases. GraphDB stays compliant with the protocol specification and allows querying data with standard HTTP requests. GraphDB's HTTP API lets you create, manage, and query repositories from any environment that supports HTTP requests, making it easy to incorporate GraphDB's capabilities into your own applications.

This first example uses the `curl` utility to execute a query (a URL-encoded version of the query `CONSTRUCT {?s ?p ?o} WHERE {?s ?p ?o} LIMIT 10`) with an HTTP `GET` request:

```
curl -G -H "Accept:application/x-trig" \
  -d query=CONSTRUCT+%7B%3Fs+%3Fp+%3Fo%7D+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10 \
  http://localhost:7200/repositories/yourrepository
```

This query executes the example query with a `POST` operation:

```
curl -X POST --data-binary @file.sparql -H "Accept: application/rdf+xml" \
  -H "Content-type: application/x-www-form-urlencoded" \
  http://localhost:7200/repositories/yourrepository
```

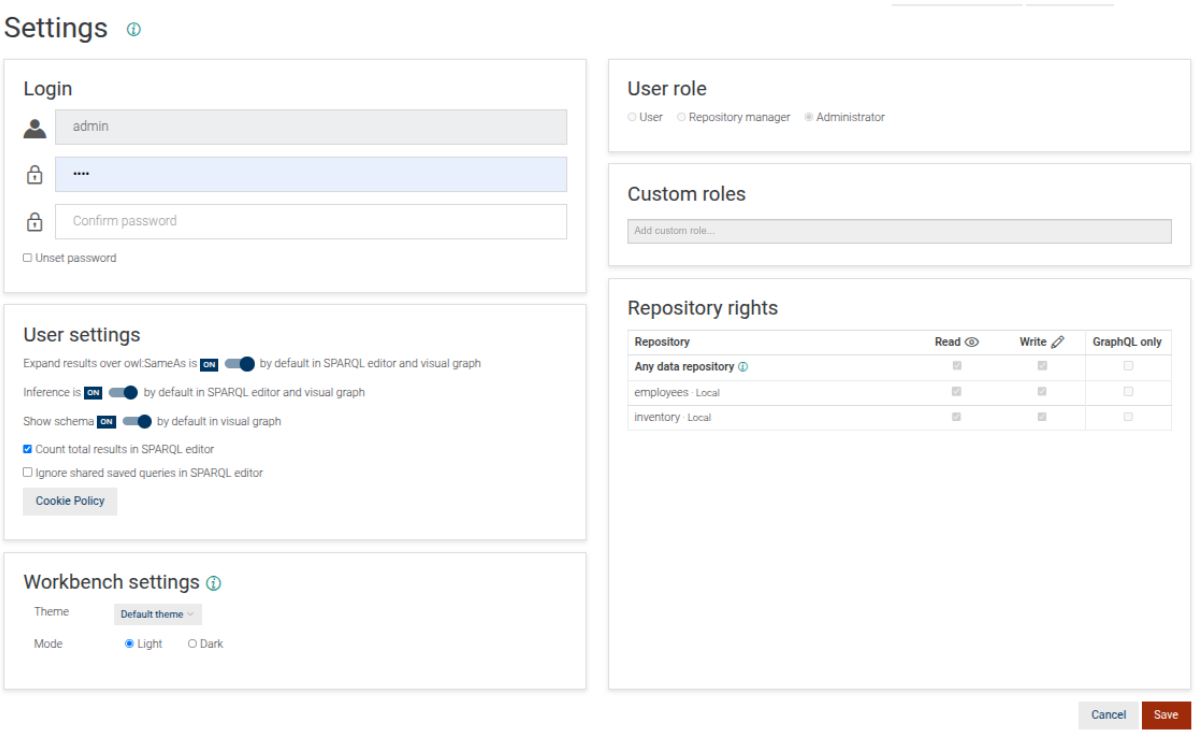
In that example, `file.sparql` contains this encoded query:

```
query=CONSTRUCT+%7B%3Fs+%3Fp+%3Fo%7D+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10
```

Tip: See *GraphDB APIs* for more information on use the GraphDB APIs.

1.3.3 Customizing Workbench behavior

Selecting *Setup* ▸ *My Settings* displays a screen that lets you customize Workbench behavior.



Note:

- Changing values on this screen requires you to enter the account password. If you have not created a new user account or changed the password for the default admin user, its password is "root".
- Workbench uses a similar screen when admin users configure user accounts. See *User management* for descriptions of the *User role*, *Custom role*, and *Repository rights* sections of this screen.

1.3.3.1 User settings

The GraphDB application settings are user-based. When security is ON, each user can access their own settings through the *Setup ▸ My Settings* menu. The admin user can also change other users' settings by selecting *Setup ▸ Users and access* and then clicking the "Edit user" pencil icon on the row of a particular user account.

When security is OFF, the settings are global for the application and available through *Setup ▸ My Settings*.

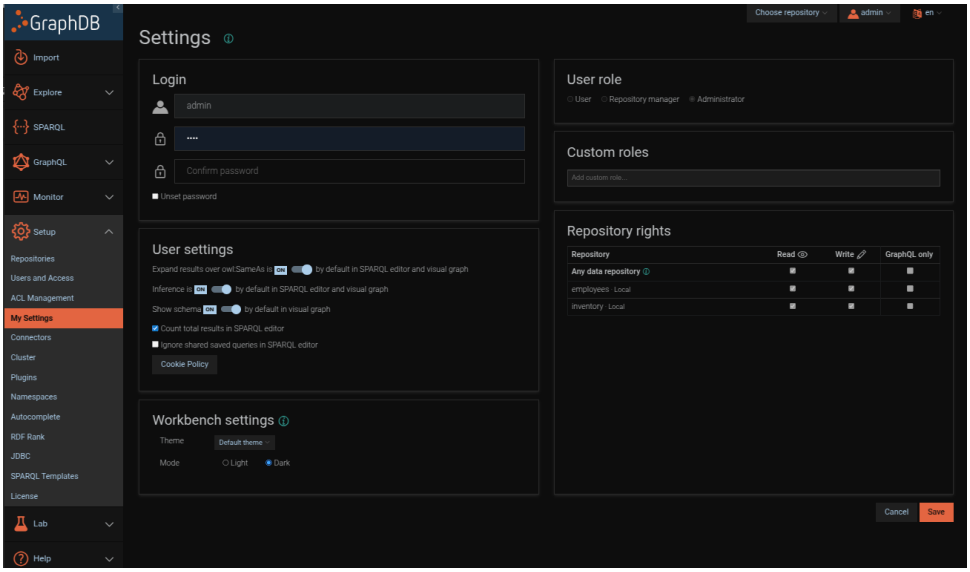
When free access is ON, only the admin can edit the *Free Access configuration*, which applies to the anonymous user.

Setting name	Description
Expand results over owl:sameAs	The default value for the *Expand results over owl:sameAs* option in the SPARQL editor. This value is used each time a new tab is created. Note that once you toggle the value in the editor, the changed value is saved in your browser, so the default is used only for new tabs. The setting is also reflected in the *Graph settings* panel of the Visual graph.
Inference	Same as above, but for the *Include inferred data in results* option in the SPARQL editor. The setting is also reflected in the *Graph settings* panel of the Visual graph.
Show schema by default in visual graph	Include or exclude predicates from the namespaces `owl:`, `rdf:`, `rdfs:`, `sesame:`, `dul:`, `prov:`, `fibo:`, and `wd:`.
Count total results in SPARQL editor	For each query without limit sent through the SPARQL editor, send an additional query to determine the total number of results. This value is useful both for your information and for results pagination. In some cases, for performance reasons, you may not want this additional query to be executed. You can set this option to `false` to prevent the execution of the additional query.
Ignore shared saved queries in SPARQL editor	In the SPARQL editor, saved queries can be shared, and you can choose not to see them.

1.3.3.2 Workbench settings

Note: The settings included in this section are stored in the browser, and not with the user's account.

You can change the appearance of Workbench by changing the Color theme or by switching between *Light* and *Dark* modes regardless of the theme you are currently using. The following shows the *Settings* screen in dark mode:

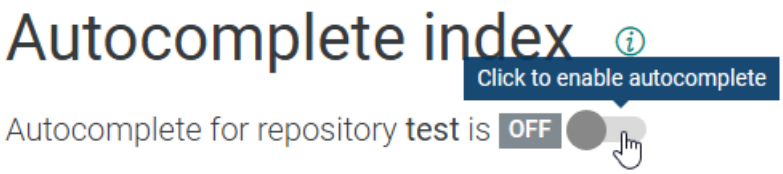


GraphDB comes with two built-in theme you can switch between — the Default GraphDB theme, and the Ontotext original theme. Workbench also allows color themes to be provided by developers and implemented as plugins. For more information on how to develop your own color themes, check the developer guide documentation on the Workbench project github repository.

1.3.4 Autocomplete index

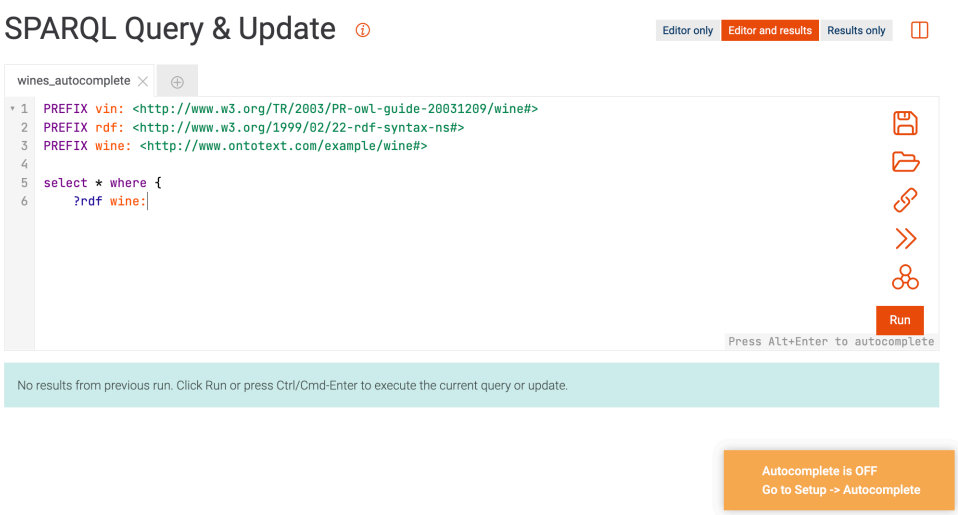
The Autocomplete index offers suggestions for local IRI names in the SPARQL editor, the *View Resource* page, and in the *Search RDF resources* box. It is an open-source GraphDB plugin that builds an index over all IRIs in the repository plus some additional well-known IRIs from RDF4J vocabularies.

The index is disabled by default. In the Workbench, you can enable it from *Setup* ▸ *Autocomplete*.



Note: If you are getting peculiar results and you think the index might be broken, use the Autocomplete screen's *Build Now* button to rebuild the index.

If you try to use autocompletion before it is enabled, a tooltip will warn you that the index is off and provide a link for building it.



You can also enable it with a *SPARQL query* from the Workbench SPARQL editor.

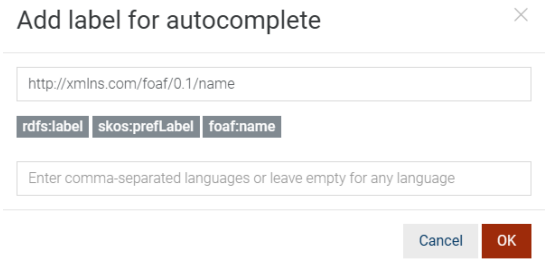
1.3.4.1 How the index works

All IRIs and their labels are split into words (tokens). During search, the whole words or their beginnings are matched.

For each IRI, the index includes the following:

- The text of the IRI local name is tokenized
- If the IRI is part of a triple `<IRI rdfs:label ?label>`, the text of the label literal is tokenized and indexed
- If the IRI is part of a triple `<IRI ?p ?label>`, and `?p` is added to the index configuration as a label predicate, then the text of the `?label` is tokenized and indexed for this IRI

You can add a new label via the *Add label* button on the right in the *Autocomplete* screen. You can also edit a label's languages by clicking the button. Both of these buttons will open this dialog box:



Local name tokenization

Local names are split by special characters such as _ and - or in cases when they contain camelCase or numbers. For example:

IRI	Local name tokens
`http://dbpedia.org/resource/Bulgarian_Tournament_Cup`	Bulgarian Tournament Cup
`http://dbpedia.org/resource/Post-rock`	Post rock
`https://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#ChardonnayGrape`	Chardonnay Grape
`https://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#USRegion`	US Region
`https://purl.org/dc/terms/ISO639-3`	ISO 639 3

Search strings

You can search for one or more words. When searching for multiple words, they can be separated with space, or with `-` and `_` symbols, in which case these will be required to be present in the matched text as well. You can also use camelCase notation to split the search string into multiple words.

Once the search string has been split into words, search is case-insensitive. When typing multiple words, each of them is treated as full match search and must be fully typed except for the last one, which is treated as *startsWith*. The order of the search string words is irrelevant, e.g., `whiteWin` would return the same results as `wineWhit`.

Some examples:

Search string	Found IRI
"Tour"	http://dbpedia.org/resource/Bulgarian_**Tour**nament_Cup
"white win" OR "whiteWin"	http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#**WhiteWin**e http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#**White**Table**Win**e
"uk-wal"	https://www.bbc.com/news/**uk-wal**es-44849196
"63"	http://purl.org/dc/terms/ISO**63**9-3

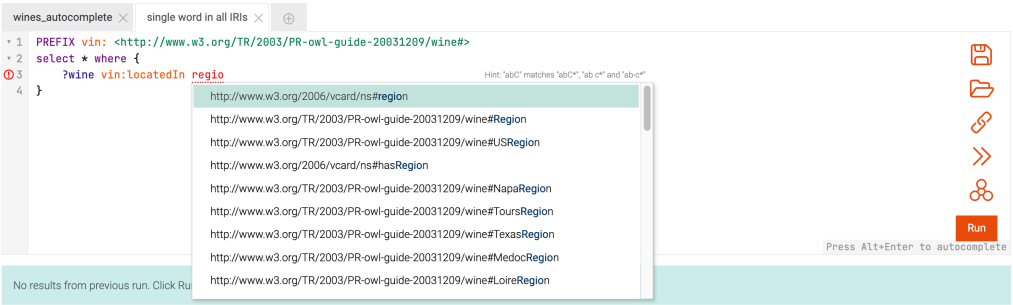
1.3.4.2 Autocomplete in the SPARQL editor

For the examples below, we will be using the W3C wine ontology dataset that you can import in your repository.

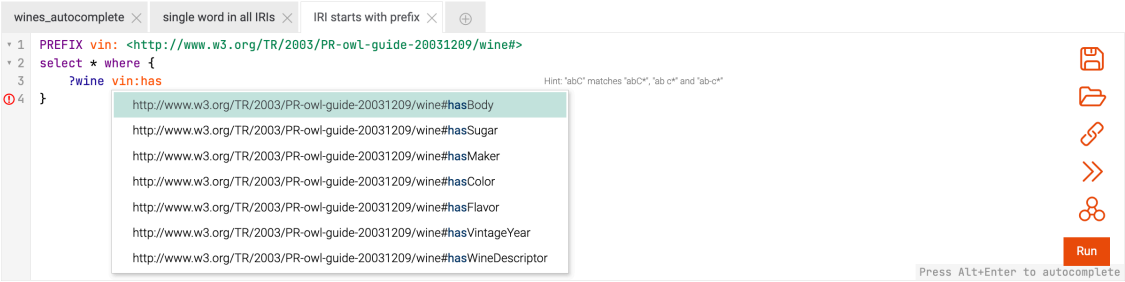
To start autocompletion in the SPARQL editor, use the shortcuts `Alt+Enter`, `Ctrl+Space`, or `Cmd+Space`, depending on your operating system and the way you have set up your shortcuts.

You can use autocompletion to:

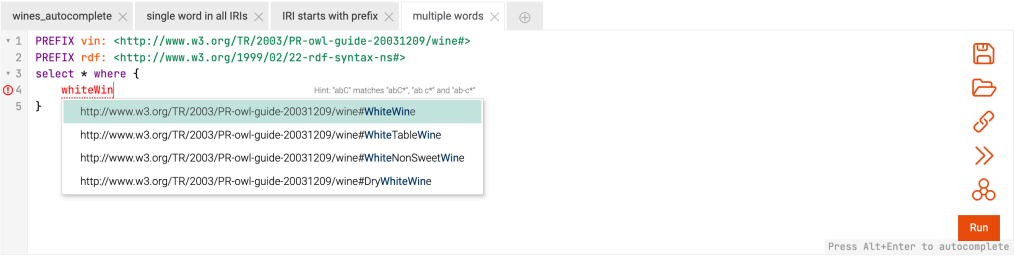
- Search for a single word in all IRIs:



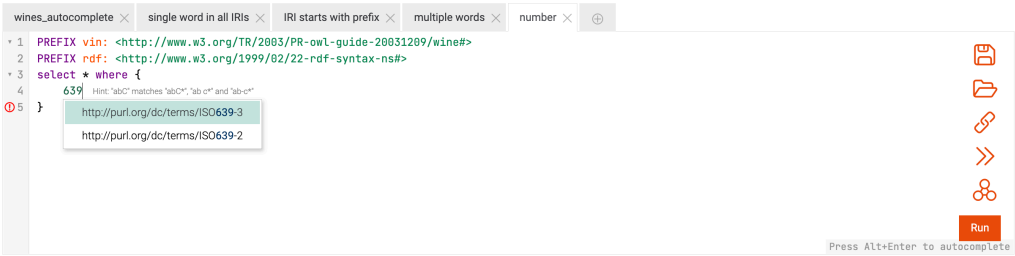
- Search only for IRIs that start with a certain prefix:



- Search for more than one word:

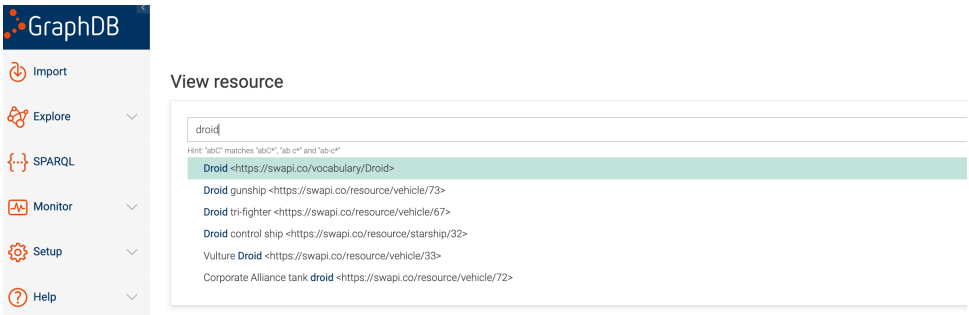


- Search by number, since indexed text is split where digits or digit sequences are found:



1.3.4.3 Autocomplete in the View resource box

To use the autocompletion feature to find a resource, go to the GraphDB home page and start typing in the *View resource* field.



You can also autocomplete resources in the *Search RDF resource* box, which is visible in all GraphDB screens in the top right corner and works the same way as the *View resource* field in the home page. Clicking the icon will open a search field where you can explore the resources in the repository.

1.3.4.4 Using SPARQL queries to work with the autocomplete index

You can also work with the autocomplete index via SPARQL queries in the Workbench SPARQL editor. Some important examples:

- Check if the index is enabled

```
ASK WHERE {
    _:s <http://www.ontotext.com/plugins/autocomplete#enabled> ?o .
}
```

- Enable the index

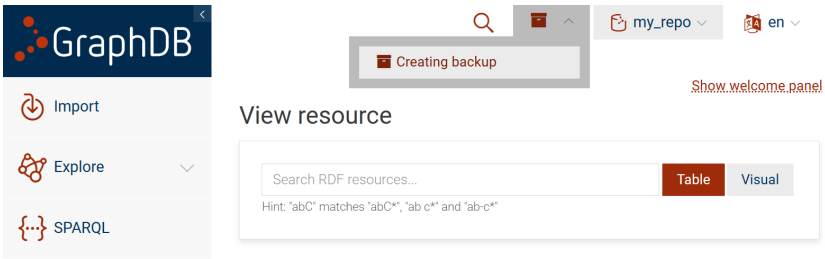
```
INSERT DATA {
    _:s  <http://www.ontotext.com/plugins/autocomplete#enabled> true .
}
```

- Autocomplete IRIs (here with the *wines* example from earlier)

```
SELECT ?s WHERE {
    ?s <http://www.ontotext.com/plugins/autocomplete#query> "win"
}
```

1.3.5 The Workbench notification area

Every Workbench view has a global notification area on the top next to the repository selection menu. The area displays various statuses and notifications related to cluster status, backup and restore operations, running queries and updates, and Workbench imports.



Icons in the notification area are color-coded:

Blue (informational) Represents operations that do not affect the performance of your repository or cluster, such as running queries.

Dark orange (warning) Represents a transient drop in performance, such as when a backup is being created or a cluster node is being replicated. In this state, the system is fully writable and no action is required.

Red (critical) Represent a state where writes are not possible, such as when a backup is being restored or there are not enough cluster nodes to form a quorum. This state may or may not require some user action for faster recovery.

In addition to that, the background color of the area itself will blink in a paler orange or red color whenever any of the operations is in the warning or critical state.

Clicking on the notification area opens a drop-down menu with additional details. Clicking on individual items in that menu will open the respective monitoring view in Workbench.

The operations that can be displayed in the notification area are:

Queries and updates Notifies the user about any kind of *update or query* operations and displays their number next to the icon.

Icon	State	State color
	Running one or more queries	**Blue (informational)**
	Running one or more updates	**Blue (informational)**

Workbench imports Notifies the user about current *Workbench import* operations and displays their number next to the icon.

Icon	State	State color
	Running one or more imports	**Blue (informational)**

Backup and restore Notifies the user about any kind of *backup or restore* operation that may block other operations.

Icon	State	State color
	Creating backup	**Dark orange (warning)**
	Creating cloud backup	**Dark orange (warning)**
	Restoring backup	**Red (critical)**
	Restoring cloud backup	**Red (critical)**

Cluster state When a cluster is running, this icon's color indicates the *state of the cluster*.

Icon	State	State color
	In sync The cluster is fully operational.	**Blue (informational)**
	Recovering The majority of nodes are in sync, but there is a recovery procedure currently underway. The cluster is writable.	**Dark orange (warning)**
	Unavailable nodes The majority of nodes are available, but some nodes are unavailable. The cluster is writable.	**Dark orange (warning)**
	Out of sync The majority of the nodes are unavailable and the cluster is **NOT writable**.	**Red (critical)**

CHAPTER

TWO

MANAGING REPOSITORIES

2.1 Overview of repositories

Data in GraphDB is organized within repositories. Each repository is an independent RDF database that can be active separately from other repositories. Operations involving data updates or queries are always directed to a single repository. Multiple repositories can be active at the same time and they are isolated from each other.

Repositories are typically native GraphDB repositories. However, there are other repository types that are used with Virtualization (which allows GraphDB to work with relational databases) and FedX Federation.

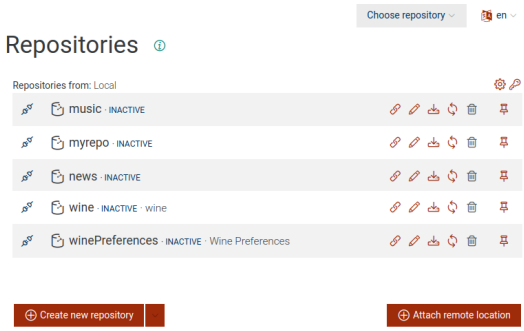
At any given time, there can be only one active repository in GraphDB. When using the Workbench, you can change the active repository from the dropdown menu on the top right in the Workbench. In addition to that, Workbench links contain a repository parameter, which will change the active repository in Workbench when loaded.

See *Creating and editing repositories* to get started using repositories.

Note: The *console* utility lets you perform many operations on repositories from your operating system command line.

2.2 Creating and editing repositories

Selecting *Repositories* from the Workbench *Setup* menu displays a list of the currently available repositories:



You can connect to one of the listed repositories by selecting its name from the *Choose repository* drop-down list in the upper right or by clicking the *Connect repository* icon to the left of its name. This will convert the icon to a *Connected repository* icon and show the repository's name in the upper right.

Mousing over the icons to the right of each repository name shows what each icon does. (Their activities are described further below.)

2.2.1 Creating a repository

Note: Having more than five repositories requires a *GraphDB Enterprise or Standard license*. All GraphDB, *Ontop*, and *FedX* repositories count toward this maximum of five. If you create a sixth one with GraphDB Free, you will see the following in the upper right of Workbench to indicate that you are in a restricted mode:

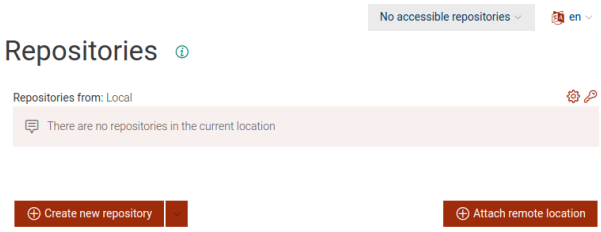
When you are in this restricted mode, the only thing you will be able to do with these repositories is *export* their data until you either upgrade to GraphDB Enterprise or delete all but five repositories.

There are two ways to create and manage repositories: through the Workbench interface or by using the RDF4J console.

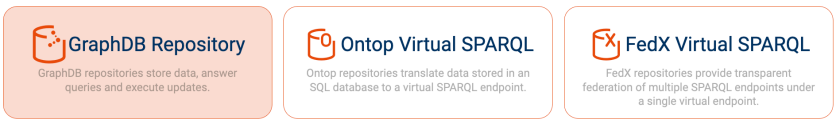
Note: To learn about the RDF4J `console` utility see the *console* section of the "GraphDB command line tools" documentation.

2.2.1.1 Using Workbench

To manage your repositories, go to *Setup* ‣ *Repositories*. This opens a list of available repositories like the one shown above; if you have just installed GraphDB, the list may be empty, as shown here:



1. Click the *Create new repository* button or create one from a file by *using the configuration template* that can be found under `/configs/templates` in the GraphDB distribution.



2. Select *GraphDB Repository* and then fill out the *Create GraphDB repository* form. The documentation page *Configure a repository through the GraphDB Workbench* describes the fields on this form.

Tip: On the *Repositories* screen shown above, if any repositories are listed, clicking the icon on the row for one of them will let you edit that repository's configuration using the same form.

3. If you have attached *remote locations* to your GraphDB instance, there will be an additional field for specifying the *Location* in which you want to create the repository. The default is Local. See more about creating a repository in a remote location a little further below.

Tip: For repositories with over several tens of millions of statements, see *the configuration parameters*.

4. Click the *Create* button. Your newly created repository appears in the repository list.

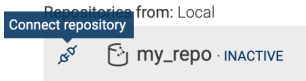
Creating a repository in a remote location

If your local copy of GraphDB is attached to a remote instance and you have write permission on that remote instance, you can create a repository there. See *Creating a repository on an attached remote GraphDB location* for more information.

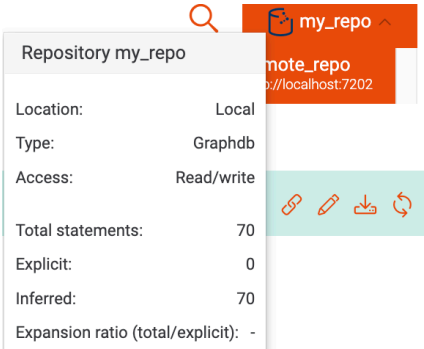
2.2.2 Using a repository

2.2.2.1 Connecting to a repository

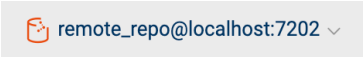
Click the *Connect repository* icon on the row showing the name of the repository that you want to connect to in the repository list.



You can also use the dropdown menu in the top right corner. This allows you to easily switch between repositories while running queries or importing and exporting data in other views. Hovering over a repository's name on the dropdown menu will also display some basic information about it.



Note: The label in the top right corner of a remote repository indicates whether you are connected to it.



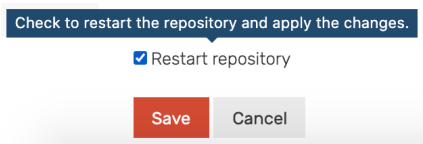
2.2.2.2 Making it the default repository

Click the pin icon on a repository's row on the *Repositories* screen to make that repository the default one that is connected each time you start up GraphDB.

2.2.2.3 Restarting a repository

There are two ways to restart a repository without having to restart the entire GraphDB instance:

- Click the *Restart repository* icon next to the repository's name on the *Repositories* screen. A warning will prompt you to confirm the action.
- Click the edit icon on the same row to open the repository configuration. At the bottom of the screen, tick the restart box, save, and confirm.



Warning: Restarting the repository will shut it down immediately, and all running queries and updates will be cancelled.

2.3 Configuring a repository

Before you start adding or changing the parameter values, we recommend planning your repository configuration and familiarizing yourself with what each of the *parameters* does, what *data structures* GraphDB supports, and what configuration values are optimal for your setup.

2.3.1 Plan a repository configuration

The following sections will help you plan your repository configuration:

- *Hardware sizing*
- *Configuration parameters*
- *GraphDB storage and indexing*
- *Configure Java heap memory*
- *Configure entity pool memory*
- *Repository Caching Mechanism*

2.3.2 Configure a repository through the GraphDB Workbench

This section describes the fields on the configuration form that you use to create a new repository and to edit the settings of an existing repository. To edit the settings of an existing one, click the *Edit repository* icon on the *Repositories* screen.

- *Repository ID*: Unique identifier for the repository. The ID must contain only letters (a-z, A-Z), numbers (0-9), and the characters '-' and '_'. This is a required field when creating a new repository. When editing the configuration of an existing one, click the edit icon shown with this field to change the repository ID value.)
- *Repository description*: Human readable description of the repository.
- *Read-only*: Prevents any edits to the repository's data.

Inference and Validation

- *Ruleset*: Sets of axiomatic triples, consistency checks and entailment rules, that determine the applied semantics with a PIE file. When creating a repository, a *Custom ruleset* button with this field lets you upload a *custom ruleset*.
  - *Disable owl:sameAs*: GraphDB uses a *non-rule implementation* of `owl:sameAs` that can be enabled or disabled independently of the ruleset.
  - *Enable consistency checks*: Enables *Consistency checks* with the ruleset.
  - *Enable SHACL validation*: Enables support for *SHACL validation* in this repository. If selected, a *SHACL options* link displays additional options, described at *Creating and configuring a SHACL repository*, that let you customize GraphDB's use of SHACL.

Indexing

- *Entity ID size*: Select 32-bit or 40-bit to define the bit size of internal IDs used to index entities such as URIs, blank nodes, literals, and RDF* embedded triples. Use 40-bit ID only if you expect more than 2 billion unique RDF values.

  Note: Using 40-bit entity IDs requires a *GraphDB Enterprise or Standard license*.

  - *Enable context index*: Builds an additional Context-Predicate-Subject-Object index to boost the SPARQL query performance of queries with `GRAPH`, `FROM`, and `FROM NAMED` clauses.
  - *Enable predicate list index*: Enables the mappings from an entity (subject or object) to its predicates; this can significantly speed up queries that use wildcard predicate patterns.
  - *Enable full-text search (FTS) index*: Enables the use of *Full-text search* with this repository.
- *FTS indexes to build (comma delimited)*: List of languages that should have a specific index with appropriate analyzer for full-text search. Possible values: default, iri, ar, bg, bn, br, ca, ja, ko, zh, ku, cz, da, de, el, en, es, et, eu, fa, fi, fr, ga, gl, hi, hu, hy, id, it, lt, lv, ne, nl, no, pt, ro, ru, sr, sv, ta, te, th, tr.
- *FTS index for xsd:string literals*: The index in which the string literals (`xsd:string`) are indexed. Possible values are: none, default or a two-letter language code.
- *FTS index for full-text indexing of IRIs*: The index in which the IRIs are indexed for full-text search. Possible values are: none, default or a two-letter language code.

Queries and Updates

- *Query timeout (seconds)*: Sets the number of seconds after which the evaluation of a query will be terminated; values less than or equal to zero mean no limit. The *Throw exception on query timeout* checkbox with this field sets the repository to throw a `QueryEvaluationException` when the duration of a query execution exceeds the timeout parameter.
- *Limit query results*: Sets the number of seconds after which the evaluation of a query will be terminated. Values less than or equal to zero mean no limit.
- *Restart repository*: (Only displayed when editing an existing repository's configuration) checking this indicates that when you click the form's *Save* button you want the repository restarted with the new settings applied.

Note: If you need a repository with *SHACL validation* enabled , you must enable this option at configuration time. SHACL validation cannot be enabled after the repository has been created.

2.3.3 Edit a repository configuration using Workbench

Some of the parameters you specify at repository creation time can be changed at any point.

1. On the *Repositories* screen, click the *Edit* icon ✎ next to a repository to display the form that lets you edit the repository's parameters. The documentation page *Configure a repository through the GraphDB Workbench* describes the fields on this form.
2. Restart the repository for the changes to take effect.

2.3.4 Configure a repository programmatically

Tip: GraphDB uses an RDF4J configuration template to configure its repositories. RDF4J keeps the repository configurations with their parameters modeled in RDF. Therefore, in order to create a new repository, RDF4J needs such an RDF file.

To configure a new repository programmatically:

1. Fill in the `graphdb.ttl` configuration template that can be found in the `/configs/templates` directory of the GraphDB distribution. The parameters are described in the *Configuration parameters* section. Here is an example:

```
# Example RDF4J configuration template for a GraphDB repository named "wines"

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rep: <http://www.openrdf.org/config/repository#> .
@prefix sail: <http://www.openrdf.org/config/sail#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#wines> a rep:Repository;
    rep:repositoryID "wines";
    rep:repositoryImpl [
        rep:repositoryType "graphdb:SailRepository";
        <http://www.openrdf.org/config/repository/sail#sailImpl> [
            <http://www.ontotext.com/config/graphdb#base-URL> "http://example.org/owlim#
↪";
            <http://www.ontotext.com/config/graphdb#check-for-inconsistencies> "false";
            <http://www.ontotext.com/config/graphdb#defaultNS> "";
            <http://www.ontotext.com/config/graphdb#disable-sameAs> "true";
            <http://www.ontotext.com/config/graphdb#enable-context-index> "false";
            <http://www.ontotext.com/config/graphdb#enable-fts-index> "false";
            <http://www.ontotext.com/config/graphdb#enable-literal-index> "true";
            <http://www.ontotext.com/config/graphdb#enablePredicateList> "true";
            <http://www.ontotext.com/config/graphdb#entity-id-size> "32";
            <http://www.ontotext.com/config/graphdb#entity-index-size> "10000000";
            <http://www.ontotext.com/config/graphdb#fts-indexes> ("default" "iri");
            <http://www.ontotext.com/config/graphdb#fts-iris-index> "none";
            <http://www.ontotext.com/config/graphdb#fts-string-literals-index> "default
↪";
            <http://www.ontotext.com/config/graphdb#imports> "";
            <http://www.ontotext.com/config/graphdb#in-memory-literal-properties> "true
↪";
            <http://www.ontotext.com/config/graphdb#query-limit-results> "0";
            <http://www.ontotext.com/config/graphdb#query-timeout> "0";
            <http://www.ontotext.com/config/graphdb#read-only> "false";
            <http://www.ontotext.com/config/graphdb#repository-type> "file-repository";
            <http://www.ontotext.com/config/graphdb#ruleset> "rdfsplus-optimized";
            <http://www.ontotext.com/config/graphdb#storage-folder> "storage";
            <http://www.ontotext.com/config/graphdb#throw-QueryEvaluationException-on-
↪timeout>
```

(continues on next page)

(continued from previous page)

```
            "false";
            sail:sailType "graphdb:Sail"
        ]
    ];
    rdfs:label "" .
```

2. Use the RDF4J API to load the configuration. See *Create a repository* for more information.

2.3.5 Configuration parameters

This is a list of all repository configuration parameters. Some of the parameters **can be changed** (effective after a restart), some **cannot be changed** (the change has no effect) and others **need special attention** once a repository has been created, as changing them will likely lead to inconsistent data (for example, unsupported inferred statements, missing inferred statements, or inferred statements that cannot be deleted).

Parameter name	Description	Default value
`base-URL`	Specifies the default namespace for the main persistence file. Non-empty namespaces are recommended, because their use guarantees the uniqueness of the anonymous nodes that may appear within the repository.	`none` Can be changed.
`check-for-inconsistencies`	Enables or disables the mechanism for consistency checking. If this parameter is `true`, consistency checks are defined in the rule file and applied at the end of every transaction. If an inconsistency occurs while committing a transaction, the whole transaction will be rolled back. See *Consistency checks* for more information.	`false` Can be changed.
`defaultNS`	Default namespaces corresponding to each imported schema file, separated by semicolons. The number of namespaces must be equal to the number of schema files from the imports parameter described below. Example: `graphdb:defaultNS "http://www.w3.org/2002/07/owl#;http://example.org/"` **Warning:** This parameter cannot be set via a command line argument.	`<empty>` Cannot be changed.
`disable-sameAs`	Enables or disables the `owl:sameAs` optimization. See *Optimization of owl:sameAs* for more information.	`true` Can change in the UI depending
`enable-context-index`	Possible value: `true`, where GraphDB will build and use the context index. See *Context index* for more information.	`false` Can be changed.
`enable-literal-index`	Enables or disables the use of a *literal index*. The literal index is always built as data is loaded or modified. This parameter only affects whether the index is used during query answering. See *Literal index* for more information.	`true` Can be changed.
`enablePredicateList`	Enables or disables mappings from an entity (subject or object) to its predicates; enabling it can significantly speed up queries that use wildcard predicate patterns. See *Predicate lists* for more information.	`true` Can be changed.
`enable-fts-index`	Enables or disables the full-text search index. In general, searching is performed via SPARQL queries that use a pattern like this: `?value onto:fts (query index limit)` See *Simple full-text search index*	`false` Can be changed.

2.3.6 Namespaces defined for the repository

Under *Setup* ▸ *Namespaces* in the GraphDB Workbench you can view and manipulate the RDF namespaces and prefixes for the active repository. Each GraphDB repository contains the following predefined prefixes:

Prefix	IRI	Description
gn:	http://www.geonames.org/ontology#	The GeoNames Ontology makes it possible to add geospatial semantic information to the Word Wide Web. Over 11 million geonames toponyms have a unique URL with a corresponding RDF web service.
owl:	http://www.w3.org/2002/07/owl#	As with RDFS, properties in OWL are used to link things together. OWL provides a rich and complex vocabulary for saying things about these links. It allows you to construct some fairly complex, but useful, relationships among classes.
path:	http://www.ontotext.com/path#	GraphDB extends SPARQL with the Graph Path Search features that allow you to not only find complex relationships between resources but also explore them and use them as filters to identify graph patterns.
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#	The RDF Schema for the RDF vocabulary terms in the RDF Namespace, defined in RDF 1.1 Concepts and Abstract Syntax.
rdfs:	http://www.w3.org/2000/01/rdf-schema#	A general-purpose language for representing simple RDF vocabularies on the Web, RDF Schema is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources.
wgs:	http://www.w3.org/2003/01/geo/wgs84_pos#	GraphDB provides support for two-dimensional geospatial data that uses the WGS84 Geo Positioning RDF vocabulary (World Geodetic System 1984).
xsd:	http://www.w3.org/2001/XMLSchema#	The standard set of datatypes available to RDF.
fn:	http://www.w3.org/2005/xpath-functions	XPATH functions on datatypes.
ofn:	http://www.ontotext.com/sparql/functions/	Beside the standard SPARQL functions operating on numbers, GraphDB offers several additional functions, allowing users to do more mathematical operations.
spif:	http://spinrdf.org/spif#	SPIN provides meta-modeling capabilities that allow users to define their own SPARQL functions and query templates. SPIN makes a library of common functions available.
afn:	http://jena.apache.org/ARQ/function#	GraphDB supports Jena simple functions analogs.
list:	ttp://jena.apache.org/ARQ/list#	GraphDB supports Jena list functions analogs.
agg:	http://jena.apache.org/ARQ/function/aggregate#	GraphDB supports Jena aggregate functions analogs.
apf:	http://jena.apache.org/ARQ/property#	GraphDB supports Jena property functions analogs.
geof:	http://www.opengis.net/def/function/geosparql/	GeoSPARQL defines a vocabulary for representing geospatial data in RDF, and it defines an extension to the SPARQL query language for processing geospatial data.
geoext:	http://rdf.useekm.com/ext#	On top of the standard GeoSPARQL functions, GraphDB adds a few useful extensions based on the USeekM library.
omgeo:	http://www.ontotext.com/owlim/geo#	Used by an Ontotext geospatial extension.
math:	http://www.w3.org/2005/xpath-functions/math	XPATH namespace used for some mathematical functions.
map:	http://www.w3.org/2005/xpath-functions/map	XPATH namespace used for some functions that manipulate maps.
array:	http://www.w3.org/2005/xpath-functions/array	XPATH namespace used for some functions that manipulate arrays.
rep:	http://www.openrdf.org/config/repository#	Parameter namespace for an RDF4J repository configuration consisting of a single RDF subject of type `rep:Repository`.
sr:	http://www.openrdf.org/config/	Parameter namespace for an RDF4J `SailRepository` repository implementation, which is the main implementation for direct access to

2.3.7 Reconfigure a repository

Once a repository is created, you can change some *parameters*, either by editing it in the Workbench or by setting a global override for a given property.

Note: When you change a repository parameter, you need to restart GraphDB for the changes to take effect.

2.3.7.1 Using Workbench

To edit a repository parameter in the GraphDB Workbench, go to *Setup* ▸ *Repositories* and click the *Edit* icon for the repository whose *parameters* you want to edit.

2.3.7.2 Global overrides

It is also possible to override a repository parameter for all repositories by setting a configuration or system property. See *Engine properties* for more details.

2.3.8 Rename a repository

2.3.8.1 Using Workbench

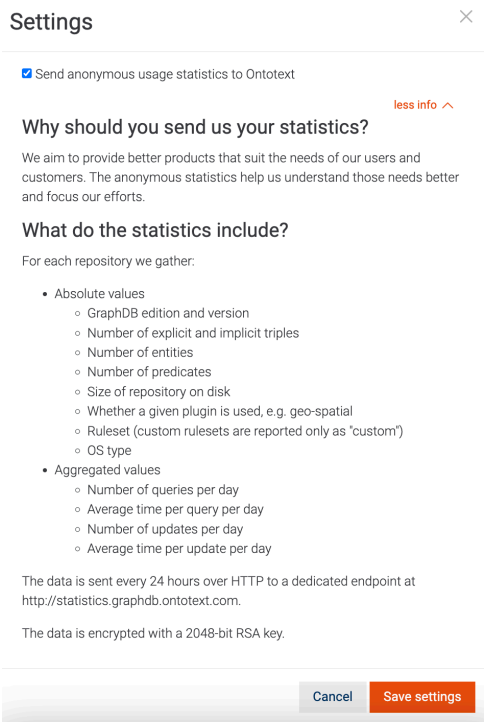
You can rename a repository on the screen where you *edit the repository configuration*. This will update all locations in the Workbench where the repository name is used.

2.3.9 Change location settings

You can enable or disable the sending of anonymous statistics to Ontotext. The default setting depends on the GraphDB license that you are using. With GraphDB Free, it is enabled by default, and with GraphDB Standard and Enterprise, it is disabled by default.

To enable or disable it manually, click the *Repository* screen's gear icon that has the tooltip *Edit common settings for these repositories*.

This displays the *Settings* dialog box, which includes a "more info" link that adds additional information to the dialog box as shown here:



2.3.10 Repository Caching Mechanism

You can also automatically limit the number of active repositories in your GraphDB instance through the Repository Caching Mechanism. The Repository Caching Mechanism will automatically shut down repositories based on an LRU (least-recently-used) policy. The mechanism can be configured on a timeout basis (time elapsed since last access to the repository) or a quantity basis (limiting the number of active repositories). Both of those can be enabled and configured individually of each other.

Once a repository has been shut down, it can be activated again as usual.

Note:

- The Repository Caching Mechanism is defined on a system level and is the same for all repositories. For more information, see *the System properties documentation*.
- The mechanism will never shut down repositories with active repository connections. If more repositories are active than the caching size permits it will wait until a repository with 0 active connections becomes available to be shut down.

2.3.10.1 Limiting the number of active repositories

You can control the number of active repositories using the `graphdb.max.active.repositories` system property. By default, GraphDB has a cache limit of 100 active repositories. Once the limit is reached, the Repository Caching Mechanism will start shutting down the least recently used repositories until all of them can fit in the cache.

To disable this policy, set the value of `graphdb.max.active.repositories` to `0`.

2.3.10.2 Set a timeout for active repositories

The Repository Caching Mechanism can start shutting down repositories that have not been accessed in a defined timeout period. By default the time policy is disabled with a default value of `0`.

The timeout can be defined through the `graphdb.repository.expiry.after.minutes` system property, which takes a value in minutes.

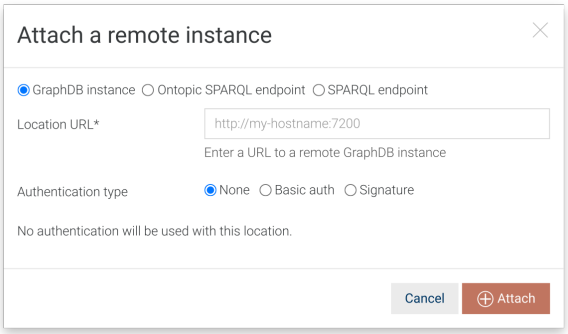
2.4 Connecting to remote locations and data sources

You can connect to remote locations and data sources from your instance of GraphDB. Locations represent individual servers or endpoints which store data. They can be other GraphDB instances or remote endpoints, and can be attached, edited, and detached.

Remote locations are mainly used for:

- Accessing remote GraphDB repositories from the same Workbench
- Accessing secured remote repositories via *SPARQL federation*

The interface for attaching to remote locations can be opened by navigating to *Setup* ▸ *Repositories* and clicking the *Attach remote location* button. This will bring up the interface for attaching to a remote locations.



Remote GraphDB instance	1. Provide the URL address of the GraphDB instance you want to attach. 2. Choose an authentication method: • *None*: The security of the remote location is disabled, and no authentication is needed. • *Basic authentication*: The security of the remote location has *basic authentication* enabled (default setting). Requires a username and a password. • *Signature*: Uses the token secret, which must be the same on both GraphDB instances. For more information on configuring the token secret, see the *GDB authentication* section of the Access Control documentation. 3. Click on *Attach*.
Ontopic SPARQL endpoint	1. Provide the URL address of the Ontopic SPARQL endpoint you want to attach. 2. Provide *ClientID* and *Secret* for authentication purposes. 3. Click on *Attach*.
SPARQL endpoint	1. Provide the URL address of the SPARQL endpoint you want to attach. 2. Provide *Username* and *Password* for authentication. 3. Click on *Attach*.

Attached remote locations appear in your list of repositories and data sources separately from the list of local repositories — *Remote GraphDB instances* lists all available repositories on attached remote GraphDB instances, while *Remote SPARQL endpoints* lists both attached SPARQL endpoints and attached Ontopic SPARQL endpoints.

Once a remote location has been attached, it can also be removed.

- For GraphDB locations, click the delete icon in the upper-right corner of the list of repositories available on that location.
- For remote SPARQL and Ontopic SPARQL endpoints, click the delete icon next to the endpoint you want to remove.

Note: The delete icon won't appear unless you have *Administrator access* to that remote location.

To change the credentials of an attached remote location, click on the edit icon.

Tip: To check the state of your current license, click the *Repository* screen's key icon that has the tooltip "View or update license for these repositories".

Note: You can connect to a remote location over HTTPS as well. To do so:

1. Enable HTTPS on the remote host.
2. Set the correct Location URL, for example `https://localhost:8083`.
3. If the certificate of the remote host is self-signed, you should add it to your JVM's Truststore. Read more about using Truststores with GraphDB in the *Use self-signed certificates* section of the documentation's *Encryption* page.

2.4.1 Creating a repository on an attached remote GraphDB location

If you have proper write permissions for an attached remote GraphDB instance, you can create a repository on that instance the same way you create a local one. From *Setup* ▸ *Repositories* ▸ *Create new repository* you can choose the remote location from the *Location* field.

2.5 Using plugins

Multiple GraphDB features are implemented as plugins based on the *GraphDB Plugin API*. Each is described in a section of the GraphDB documentation that is most relevant to the features that the plugin enables.

Plugin	Description
Semantic similarity searches	Exploring and searching semantic similarity in RDF resources.
Ranking entities by interconnectedness with the RDF Rank Plugin	Identifying the more important or more popular entities in the repository by examining their interconnectedness.
JavaScript functions	Defining and executing JavaScript code, further enhancing data manipulation with SPARQL.
Change tracking	Tracking changes within the context of a transaction identified by a unique ID.
Query time inference with the Provenance plugin	Generation of inference closure from a specific named graph at query time.
Checking inference derivation with the Proof plugin	Finding out how a given statement has been derived by the inferencer.
Graph path search	Exploring complex relationships between resources.

Several of the plugins let you create and access **user-defined indexes**. These are created with SPARQL and differ from the system indexes in that they can be configured dynamically at runtime. Any user with write access to a given repository can define such an index.

These are:

Plugin	Description
Autocomplete index	Suggestions for the IRIs' local names in the SPARQL editor and the View Resource page.
GeoSPARQL support	Support for GeoSPARQL, a standard for representing and querying geospatial linked data for the Semantic Web from the Open Geospatial Consortium (OGC). The plugin allows the conversion of Well-Known Text from different coordinate reference systems (CRS) into the CRS84 format, which is the default CRS according to the OGC.
Geospatial extensions	Support of 2-dimensional geospatial data that uses the WGS84 Geo Positioning RDF vocabulary (World Geodetic System 1984).
Data history and versioning	Accessing past states of your database through versioning of the RDF data model level.
Text mining plugin	Calling of text mining algorithms and generation of new relationships between entities.
Generating sequential IDs with the Sequences plugin	Providing transactional sequences for GraphDB. A sequence is a long counter that can be atomically incremented in a transaction to provide incremental IDs.

The *GraphDB Connectors* are also user-defined indexes.

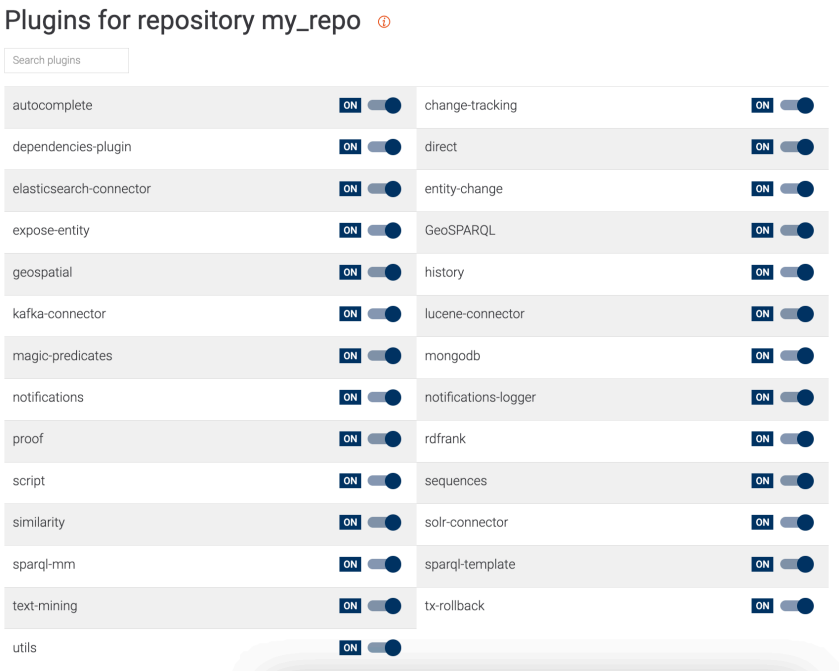
2.5.1 Activate or deactivate plugins

GraphDB plugins can be in an active or inactive state. This means attaching them to or detaching them from GraphDB on a fundamental level.

2.5.1.1 From the Workbench

Most of the plugins can be activated or deactivated from the Workbench in *Setup* ‣ *Plugins*. By default, all plugins available there are activated.

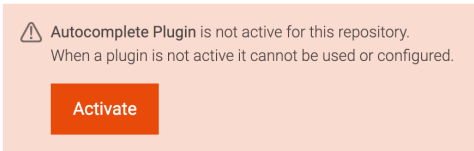
Plugins are activated or deactivated for individual repositories. For example, you might have the change-tracking plugin active for one repository and inactive for another. In the example below, all plugins are on for the *my_repo* repository.



Note: The Provenance plugin needs to be *registered* first in order to be activated. Once registered, it will appear in the list.

If you deactivate a plugin, you will not be able to *enable* it. For example:

1. In *Setup* ▸ *Plugins*, deactivate Autocomplete.
2. If you go to *Setup* ▸ *Autocomplete*, you will get the following error message:



2.5.1.2 With a SPARQL query

To activate a plugin with a query from the SPARQL editor, run:

```
INSERT DATA { <u:a> <http://www.ontotext.com/owlim/system#startplugin> "plugin-name".}
```

To deactivate it:

```
INSERT DATA { <u:a> <http://www.ontotext.com/owlim/system#stopplugin> "plugin-name".}
```

The subject of these triple patterns can be anything. The important thing is to use the predicate shown and to name the plugin to toggle.

Note: Spell out the plugin names the way they are displayed in the Workbench page shown above.

2.5.1.3 List plugins and their state

To get a list of all plugins and their current state (active/inactive), run:

```
SELECT ?plugin ?state  { ?plugin <http://www.ontotext.com/owlim/system#listplugins> ?state .}
```

2.5.2 Enable or disable plugins

Some of the plugins also have an enabled and disabled state, provided that they have been activated before that. These include:

2.5.2.1 Autocomplete index

The index can be enabled both from *Setup* ▸ *Autocomplete* in the Workbench and with a *SPARQL query*.

2.5.2.2 Change tracking

You can enable change tracking for a certain transaction ID. See how in the *Usage* section of the *Change tracking* page.

2.5.2.3 Data history and versioning

See how to enable or disable this plugin at *Enable or disable a plugin*.

2.5.2.4 GeoSPARQL support

See how to enable this plugin in the *Enable plugin* section of the *GeoSPARQL support* page.

2.5.2.5 RDF Rank

This refers to the RDF Rank filtered mode when you want to calculate the rank only of certain entities. Learn more about this in the *Rank filtering* section of the *Ranking results* page.

2.6 Inferencing

Inference is the derivation of new knowledge from existing knowledge and axioms. In an RDF database such as GraphDB, inference is used for deducing further knowledge based on existing RDF data and a formal set of inference rules.

2.6.1 Inference in GraphDB

GraphDB supports inference out of the box and provides updates to inferred facts automatically. Facts change all the time and the amount of resources it would take to manually manage updates or rerun the inferencing process would be overwhelming without this capability. This results in improved query speed, data availability and accurate analysis.

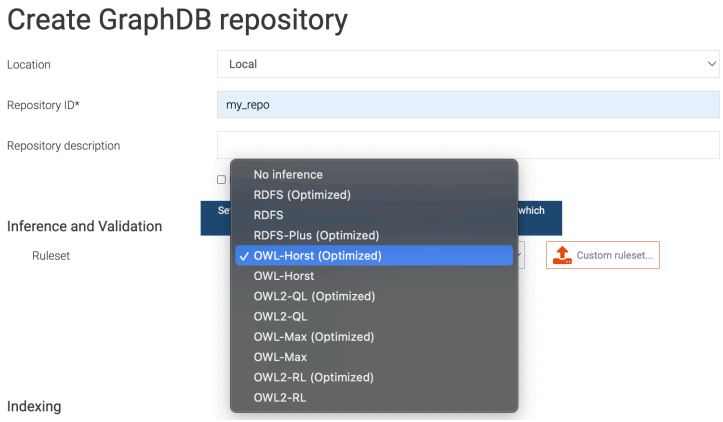
Inference uncovers the full power of data modeled with RDF(S) and ontologies. GraphDB will use the data and the rules to infer more facts and thus produce a richer dataset than the one you started with.

GraphDB can be configured via “rulesets” — sets of axiomatic triples and entailment rules — that determine the applied semantics. GraphDB relies on a compile stage, during which the rules are compiled into Java source code that is then further compiled into Java bytecode and merged together with the inference engine.

2.6.1.1 Standard rulesets

The GraphDB inference engine provides full standard-compliant reasoning for RDFS, OWL-Horst, OWL2-RL, and OWL2-QL.

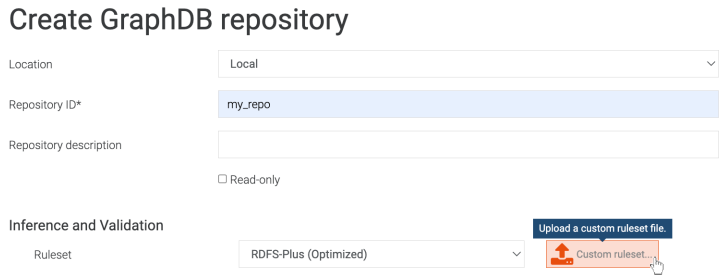
To apply a ruleset, select one from the options in the pull-down list when configuring your repository as shown below through GraphDB Workbench:



2.6.1.2 Custom rulesets

GraphDB also supports *Custom rulesets* that enable custom reasoning through the same performance-optimized inference engine. The rulesets are defined via `.pie` files. (Read more about these in the *Custom rulesets* section of the *Reasoning* page.)

Use the *Custom ruleset* button to select a `.pie` file.



See how to configure the default inference value setting from the Workbench in *Customizing Workbench behavior*.

2.6.2 Checking inference derivation with the Proof plugin

The GraphDB proof plugin lets you find out how a particular statement has been derived by the *inferencer* — in other words, which *rule* fired and which premises have been matched to produce that statement.

The plugin is available as open source.

Note: When using the Proof plugin alongside *Fine-grained access control*, the plugin will obey the FGAC checks and display only statements the user has access to. In practice this means that the explanation for any given inferred statement will be available to the user only if that user has access to *all* the statements that triggered the inference.

2.6.2.1 Predicates and namespace

The plugin supports the following predicates:

- `proof:explain`: the subject will be bound to the state variable (a unique BNode, or blank node, in request scope). The object is a list with three arguments — the subject, predicate, and object of the statement to be explained.

  When the subject is bound with the ID of the state variable, the other predicates can be used to fetch a part of the current solution (rulename, subject, predicate, object, and context of the matching premise).

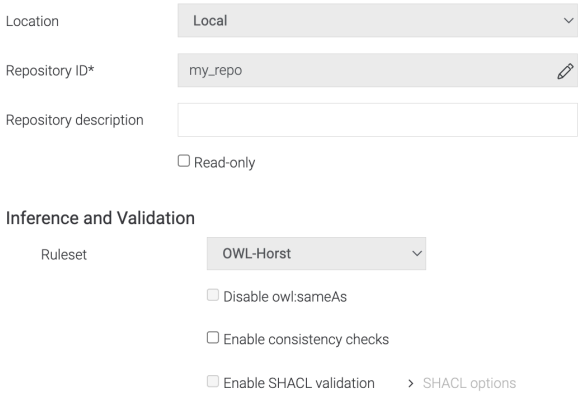
  Upon re-evaluation, values from the next premise of the rule are used, or we proceed to the next solution to enumerate its premises for each of the rules that derive the statement.

  For brevity of the results, a solution is checked for whether it contains a premise that is equal to the source statement we are exploring. If so, that solution is skipped. This removes matches for self-supporting statements (that is, when the same statement is also a premise of a rule that derives it).
- `proof:rule`: if the subject is bound to the state variable, then the current solution is accessed through the context, and the object is bound to the rule name of the current solution as a Literal. If the source statement is explicit, the Literal "explicit" is bound to the object.
- `proof:subject`: the subject is the state variable and the object is bound to the subject of the premise.
- `proof:predicate`: the subject is the state variable and the object is bound to the predicate of the premise.
- `proof:object`: the subject is the state variable and the object is bound to the object of the premise.
- `proof:context`: the subject is the state variable and the object is bound to the context of the premise (or `onto:explicit/onto:implicit`).

The plugin namespace is `http://www.ontotext.com/proof/`, and its internal name is `proof`.

2.6.2.2 Usage and examples

When creating your repository, make sure to select the OWL-Horst ruleset, as the examples below cover inferences related to the `owl:inverseOf` and `owl:intersectionOf` predicates, for which OWL-Horst contains rules.



Simple example with `owl:inverseOf`

This example will investigate the relevant rule from a ruleset supporting the `owl:inverseOf` predicate, which looks like this one in the source `.pie` file:

```
Id: owl_invOf

  a b c
  b <owl:inverseOf> d
------------------------------------
  c d a
```

Add to the repository the following data that:

- declares that `urn:childOf` is the inverse property of `urn:hasChild`
- places a statement relating `urn:John urn:childOf urn:Mary` in a context named `<urn:family>`

```
INSERT DATA {
    <urn:childOf> owl:inverseOf <urn:hasChild> .
    graph <urn:family> {
      <urn:John> <urn:childOf> <urn:Mary>
    }
}
```

The following query explains which rule has been triggered to derive the `(<urn:Mary> <urn:hasChild> <urn:John>)` statement. The arguments to the `proof:explain` predicate from the plugin are supplied by a `VALUES` expression for brevity:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX proof: <http://www.ontotext.com/proof/>
SELECT ?rule ?s ?p ?o ?context WHERE {
    VALUES (?subject ?predicate ?object) {(<urn:Mary> <urn:hasChild> <urn:John>)}
    ?ctx proof:explain (?subject ?predicate ?object) .
    ?ctx proof:rule ?rule .
    ?ctx proof:subject ?s .
    ?ctx proof:predicate ?p .
    ?ctx proof:object ?o .
    ?ctx proof:context ?context .
}
```

The result we get is:

	rule	s	p	o	context
1	"rule_owl_invOf"	urn:childOf	owl:inverseOf	urn:hasChild	http://www.ontotext.com/explicit
2	"rule_owl_invOf"	urn:John	urn:childOf	urn:Mary	urn:family

If we change the `VALUES` to:

```
VALUES (?subject ?predicate ?object) {
    (<urn:John> <urn:childOf> <urn:Mary>)
}
```

we are getting:

	rule	s	p	o	context
1	"explicit"	urn:John	urn:childOf	urn:Mary	urn:family

If we change the `VALUES` further to:

```
VALUES (?subject ?predicate ?object) {
    (<urn:hasChild> owl:inverseOf <urn:childOf>)
}
```

the result we get is:

	rule	s	p	o	context
1	"rule_owl_invOf"	owl:inverseOf	owl:inverseOf	owl:inverseOf	http://www.ontotext.com/implicit
2	"rule_owl_invOf"	urn:childOf	owl:inverseOf	urn:hasChild	http://www.ontotext.com/explicit

As you can see, `(owl:inverseOf, owl:inverseOf, owl:inverseOf)` is implicit, and we can investigate further by altering the `VALUES` to this,

```
VALUES (?subject ?predicate ?object) {
    (owl:inverseOf owl:inverseOf owl:inverseOf)
}
```

which will return the following:

	rule	s	p	o	context
1	"rule_owl_invOfBySymProp"	owl:inverseOf	rdf:type	owl:SymmetricProperty	http://www.ontotext.com/implicit

The `.pie` code for the related rule is as follows:

```
Id: owl_invOfBySymProp

  a <rdf:type> <owl:SymmetricProperty>
------------------------------------
  a <owl:inverseOf> a
```

If we track down the last premise, we will see that another rule supports it. (Note that both rules and the premises are axioms. Currently, the plugin does not check whether something is an axiom.)

```
Id: owl_SymPropByInverse

  a <owl:inverseOf> a
------------------------------------
  a <rdf:type> <owl:SymmetricProperty>
```

Example using bindings from other patterns

This more elaborate example demonstrates how to combine the bindings from regular SPARQL statement patterns and explore multiple statements.

We can define a helper JavaScript function that will return the local name of an IRI using the *JavaScript functions plugin*:

```
PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
    [] jsfn:register '''
    function lname(value) {
     if(value instanceof org.eclipse.rdf4j.model.IRI)
         return value.getLocalName();
     else
         return ""+value;
    }
'''
}
```

Next, the query will look for statements with `?subject` bound to `<urn:Mary>`, and explain all of them. Note the use of the newly defined function `lname()` in the projection expression with `concat()`:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX onto: <http://www.ontotext.com/>
prefix proof: <http://www.ontotext.com/proof/>
PREFIX jsfn: <http://www.ontotext.com/js#>
SELECT (concat('(',jsfn:lname(?subject),',',jsfn:lname(?predicate),',',jsfn:lname(?object),')') as ?
↪stmt)
    ?rule ?s ?p ?o ?context
WHERE {
    bind(<urn:Mary> as ?subject) .
    {?subject ?predicate ?object}

    ?ctx proof:explain (?subject ?predicate ?object) .
    ?ctx proof:rule ?rule .
    ?ctx proof:subject ?s .
    ?ctx proof:predicate ?p .
    ?ctx proof:object ?o .
    ?ctx proof:context ?context .
}
```

The results look as follows:

	stmt	rule	s	p	o	context
1	"(Mary,type,Resource)"	"rule_rdf1_rdfs4a_4b_2"	urn:John	urn:childOf	urn:Mary	urn:family
2	"(Mary,hasChild,John)"	"rule_owl_invOf"	urn:childOf	owl:inverseOf	urn:hasChild	http://www.ontotext.com/explicit
3	"(Mary,hasChild,John)"	"rule_owl_invOf"	urn:John	urn:childOf	urn:Mary	urn:family

The first result for `(Mary, type, Resource)` is derived from the `rdf1_rdfs4a_4b_2` rule from the OWL-Horst ruleset, which looks like the following:

```
Id: rdf1_rdfs4a_4b
    x  a  y
    -------------------------------
    x  <rdf:type>  <rdfs:Resource>
    a  <rdf:type>  <rdfs:Resource>
    y  <rdf:type>  <rdfs:Resource>
```

More complex example using other data

Let's further explore the inference engine by adding the data below into the same repository:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
INSERT data {
    <urn:Red> a <urn:Colour> .
    <urn:White> a <urn:Colour> .
    <has:color> a rdf:Property .
    <urn:WhiteThing> a owl:Restriction;
                    owl:onProperty <has:color>;
                    owl:hasValue <urn:White> .
    <urn:RedThing> a owl:Restriction;
                    owl:onProperty <has:color>;
                    owl:hasValue <urn:Red> .
    <has:component> a rdf:Property .
    <urn:Wine> a owl:Restriction;
                    owl:onProperty <has:component>;
                    owl:someValuesFrom <urn:Grape> .
```

(continues on next page)

(continued from previous page)

```
    <urn:RedWine> owl:intersectionOf (<urn:RedThing> <urn:Wine>) .
    <urn:WhiteWine> owl:intersectionOf (<urn:WhiteThing> <urn:Wine>) .
    <urn:Beer> a owl:Restriction;
                    owl:onProperty <has:component>;
                    owl:someValuesFrom <urn:Malt> .
    <urn:PilsenerMalt> a <urn:Malt> .
    <urn:PaleMalt> a <urn:Malt> .
    <urn:WheatMalt> a <urn:Malt> .

    <urn:MerloGrape> a <urn:Grape> .
    <urn:CaberneGrape> a <urn:Grape> .
    <urn:MavrudGrape> a <urn:Grape> .

    <urn:Merlo> <has:component> <urn:MerloGrape> ;
                <has:color> <urn:Red> .
}
```

It is a simple beverage ontology that uses `owl:hasValue`, `owl:someValuesFrom`, and `owl:intersectionOf` to classify instances based on the values of some of the ontology properties.

It contains:

- two colors: `Red` and `White`
- classes of `WhiteThings` and `RedThings` for the items related to `has:color` property to `White` and `Red` colors
- classes `Wine` and `Beer` for the items related to `has:component` to instances of `Grape` and `Malt` classes
- several instances of `Grape`, such as `MerloGrape` and `CabernetGrape`, and of `Malt`, such as `PilsenerMalt` and `WheatMalt`
- classes `RedWine` and `WhiteWine` are declared as intersections of `Wine` with `RedThings` or `WhiteThings` with `WhiteWine`, respectively
- an instance `Merlo` related to `has:component` with the value `MerloGrape`, and whose value for `has:color` is `Red`

The expected inference is that `Merlo` is classified as `RedWine` because it is a member of both `RedThings` (since `has:color` is related to `Red`) and `Wine` (since `has:component` points to an object that is a member of the class `Grape`).

If we evaluate:

```
DESCRIBE <urn:Merlo>
```

We will get a Turtle document as follows:

```
<urn:Merlo> a rdfs:Resource, <urn:RedThing>, <urn:RedWine>,<urn:Wine>;
     <has:color> <urn:Red>;
     <has:component> <urn:MerloGrape> .
```

As you can see, the inferencer correctly derived that `Merlo` is a member of `RedWine`.

Now, let's see how it derived this.

First, we will add some helper JavaScript functions to combine the results in more compact form as literals formed by the local names of the IRI components in the statements. We already introduced the `js:lname()` function from the *previous examples*, so we can reuse it to declare a `stmt()` function that concatenates several more items into a convenient literal:

```
PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
    [] jsfn:register '''
    function stmt(s, p, o, c) {
```

(continues on next page)

(continued from previous page)

```
     return '('+lname(s)+', '+lname(p)+', '+lname(o)+(c?', '+lname(c):'')+')';
    }
'''
}
```

We also need a way to refer to a BNode using its label because SPARQL always generates unique BNodes during query evaluation:

```
PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
    [] jsfn:register '''
    function _bnode(value) {
        return org.eclipse.rdf4j.model.impl.SimpleValueFactory.getInstance().createBNode(value);
    }
'''
}
```

Let's see how the statement (`urn:Merlo rdf:type urn:RedWine`) has been derived. Note the use of the `js:stmt()` function in the projection of the query. The query will use a SUBSELECT to provide bindings for `?subject`, `?predicate`, and `?object` variables as a convenient way to easily add more statements to be explained by the plugin further.

```
PREFIX jsfn:<http://www.ontotext.com/js#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
prefix proof: <http://www.ontotext.com/proof/>
SELECT(jsfn:stmt(?subject,?predicate,?object) as ?stmt) ?rule (jsfn:stmt(?s,?p,?o,?context) as ?premise)
WHERE {
    {
        SELECT ?subject ?predicate ?object {
            VALUES (?subject ?predicate ?object) {
                    (<urn:Merlo> rdf:type <urn:RedWine>)
            }
        }
    }
    ?ctx proof:explain (?subject ?predicate ?object) .
    ?ctx proof:rule ?rule .
    ?ctx proof:subject ?s .
    ?ctx proof:predicate ?p .
    ?ctx proof:object ?o .
    ?ctx proof:context ?context .
}
```

The result looks like this:

	stmt	rule	premise
1	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(RedWine, intersectionOf, _:node2, explicit)"
2	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(Merlo, _:allTypes, _:node2, implicit)"

The first premise is explicit, and comes from the definition of the `RedWine` class, which is an `owl:intersectionOf` of an RDF list (`_:node2`) that hold the classes that form the intersection. The second premise relates `Merlo` with a predicate called `_allTypes` to the node from the intersection node. The inference is derived after applying the following rule, where `a` is bound to `Merlo` and `c` to `RedWine`:

```
Id: owl_typeByIntersect_1

  a <onto:_allTypes> b
  c <owl:intersectionOf> b
------------------------------------
  a <rdf:type> c
```

Let's add a `(Merlo, _allTypes, _:node2)` statement to the list of statements in the SUBSELECT that we used in the query. We will change the SUBSELECT to use a UNION, where for the second part, the `?object` is bound to the right BNode that we created by using the helper `js:_bnode()` function and providing the ID as a literal:

```
SELECT ?subject ?predicate ?object {
        {
            VALUES (?subject ?predicate ?object) {
                (<urn:Merlo> rdf:type <urn:RedWine>)
            }
        } UNION {
            bind (jsfn:_bnode('node2') as ?object)
            VALUES (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>)}
        }
}
```

The results of this evaluation are:

	stmt	rule	premise
1	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(Merlo, _allTypes, _:node2, implicit)"
2	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(RedWine, intersectionOf, _:node2, explicit)"
3	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, first, RedThing, explicit)"
4	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, rest, _:node3, explicit)"
5	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, _allTypes, _:node3, implicit)"
6	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, type, RedThing, implicit)"

We can see that `(Merlo, _allTypes, _:node1)` is derived by rule `owl_typeByIntersect_3`,

```
Id: owl_typeByIntersect_3

     a <rdf:first> b
     d <rdf:type> b
     a <rdf:rest> c
     d <onto:_allTypes> c
   ------------------------------------
     d <onto:_allTypes> a
```

where we have two explicit and two inferred statements matching the premises `(Merlo, _allTypes, _:node2)` and `(Merlo, type, RedThing)`.

When we add to the list `(Merlo, type, RedThing)` first, the SUBSELECT is changed to:

```
SELECT ?subject ?predicate ?object {
        {
            VALUES (?subject ?predicate ?object) {
                (<urn:Merlo> rdf:type <urn:RedWine>)
                (<urn:Merlo> rdf:type <urn:RedThing>)
            }
        } UNION {
            bind (jsfn:_bnode('node2') as ?object)
            VALUES (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>)}
        }
}
```

The result is:

	stmt	rule	premise
1	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(Merlo, _allTypes, _:node2, implicit)"
2	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(RedWine, intersectionOf, _:node2, explicit)"
3	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, onProperty, color, explicit)"
4	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, hasValue, Red, explicit)"
5	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(Merlo, color, Red, explicit)"
6	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, first, RedThing, explicit)"
7	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, rest, _:node3, explicit)"
8	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, _allTypes, _:node3, implicit)"
9	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, type, RedThing, implicit)"

We see that `(Merlo, type, RedThing)` is derived by matching rule `owl_typeByHasVal` with all explicit premises:

```
Id: owl_typeByHasVal

  a <owl:onProperty> b
  a <owl:hasValue> c
  d b c
------------------------------------
  d <rdf:type> a
```

where `a` is bound to `RedThing` and `d` to `Merlo`, etc.

Let's add the other implicit statement that matched the `owl_typeByInterset_3` rule `(Merlo, _allTypes, _:node3)`. To do that, we will add another argument to the UNION in the SUBSELECT by introducing the `_:node2` using the same `js_bnode()` function.

The SUBSELECT looks like this:

```
SELECT ?subject ?predicate ?object {
        {
            VALUES (?subject ?predicate ?object) {
                (<urn:Merlo> rdf:type <urn:RedWine>)
                (<urn:Merlo> rdf:type <urn:RedThing>)
            }
        } UNION {
            bind (jsfn:_bnode('node2') as ?object)
            VALUES (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>) }
        } UNION {
        bind (jsfn:_bnode('node3') as ?object)
            VALUES (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>) }
        }
}
```

Evaluating it returns the following:

	stmt	rule	premise
1	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(Merlo, _allTypes, _:node2, implicit)"
2	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(RedWine, intersectionOf, _:node2, explicit)"
3	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, onProperty, color, explicit)"
4	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, hasValue, Red, explicit)"
5	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(Merlo, color, Red, explicit)"
6	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, first, RedThing, explicit)"
7	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, rest, _:node3, explicit)"
8	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, _allTypes, _:node3, implicit)"
9	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, type, RedThing, implicit)"
10	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(_:node3, first, Wine, explicit)"
11	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(_:node3, rest, nil, explicit)"
12	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(Merlo, type, Wine, implicit)"

The statement `(Merlo, _allTypes, _:node2)` was derived by `owl_typeByIntersect_2` and the only implicit statement matching as premise is `(Merlo, type, Wine)`.

The `owl_typeByIntersect_2` rule looks like this, where `c` is bound to `Merlo` and `b` to `Wine`:

```
Id: owl_typeByIntersect_2

  a <rdf:first> b
  a <rdf:rest> <rdf:nil>
  c <rdf:type> b
------------------------------------
  c <onto:_allTypes> a
```

Let's add the `(Merlo, type, Wine)` statement to the SUBSELECT we used to explore as another UNION using VALUES, and evaluate the query again:

```
SELECT ?subject ?predicate ?object {
        {
            values (?subject ?predicate ?object) {
                (<urn:Merlo> rdf:type <urn:RedWine>)
                (<urn:Merlo> rdf:type <urn:RedThing>)
            }
        } UNION {
            bind (jsfn:_bnode('node2') as ?object)
            values (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>)}
        } UNION {
 bind (jsfn:_bnode('node3') as ?object)
            VALUES (?subject ?predicate) {
                (<urn:Merlo> <http://www.ontotext.com/_allTypes>)}
        } UNION {
            values (?subject ?predicate ?object) {
                (<urn:Merlo> rdf:type <urn:Wine>)
            }
        }
}
```

The results have now been expanded, as shown in lines 13-16:

	stmt	rule	premise
1	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(Merlo, _allTypes, _:node2, implicit)"
2	"(Merlo, type, RedWine)"	"rule_owl_typeByIntersect_1"	"(RedWine, intersectionOf, _:node2, explicit)"
3	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, onProperty, color, explicit)"
4	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(RedThing, hasValue, Red, explicit)"
5	"(Merlo, type, RedThing)"	"rule_owl_typeByHasVal"	"(Merlo, color, Red, explicit)"
6	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, first, RedThing, explicit)"
7	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(_:node2, rest, _:node3, explicit)"
8	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, _allTypes, _:node3, implicit)"
9	"(Merlo, _allTypes, _:node2)"	"rule_owl_typeByIntersect_3"	"(Merlo, type, RedThing, implicit)"
10	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(_:node3, first, Wine, explicit)"
11	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(_:node3, rest, nil, explicit)"
12	"(Merlo, _allTypes, _:node3)"	"rule_owl_typeByIntersect_2"	"(Merlo, type, Wine, implicit)"
13	"(Merlo, type, Wine)"	"rule_owl_typeBySomeVal"	"(Wine, onProperty, component, explicit)"
14	"(Merlo, type, Wine)"	"rule_owl_typeBySomeVal"	"(Merlo, component, MerloGrape, explicit)"
15	"(Merlo, type, Wine)"	"rule_owl_typeBySomeVal"	"(Wine, someValuesFrom, Grape, explicit)"
16	"(Merlo, type, Wine)"	"rule_owl_typeBySomeVal"	"(MerloGrape, type, Grape, explicit)"

These come from rule `owl_typeBySomeVal` where all premises matching it were explicit. The rule looks like the following:

```
Id: owl_typeBySomeVal

  a <rdf:type> b
  c <owl:onProperty> d
  c <owl:someValuesFrom> b
  e d a
------------------------------------
  e <rdf:type> c
```

where `e` is bound to `Merlo`, `d` to `has:component`, `a` to `MerloGrape`, `b` to `Grape`, and so forth.

In conclusion, while the chain is rather obscure, we were able to find out how the inferencer derived `(<urn:Merlo> rdf:type <urn:RedWine>)`:

- `(Merlo, type, Wine)` was derived by rule `owl_typeBySomeVal` from all explicit statements.
- `(Merlo, type, RedThing)` was derived by rule `owl_typeByHasVal` from explicit statements.
- `(Merlo, _allTypes, _:node2)` was derived by rule `owl_typeByIntersect_2` with combination of some explicit and the inferred `(Merlo, type, Wine)`.
- `(Merlo, _allTypes, _:node1)` was derived by rule `owl_typeByIntersect_3` with combination of explicit and inferred `(Merlo, type, RedThing)` and `(Merlo, _allTypes, _:node2)`.
- And finally, `(Merlo, type, RedWine)` was derived by `owl_typeByIntersect_1` from explicit `(RedWine, intersectionOf, _:node1)` and inferred `(Merlo, _allTypes, _:node1)`.

2.6.3 Query time inference with the Provenance plugin

The provenance plugin lets you generate inference closure from a specific named graph at query time. This is useful in situations when you want to trace what the implicit statements generated from a specific graph are and the axiomatic triples part of the configured ruleset — that is, the ones inserted with the special predicate `sys:schemaTransaction`. For more information, check *Reasoning*.

By default, GraphDB's forward-chaining inferencer materializes all implicit statements in the default graph. Therefore, it is impossible to trace which graphs these implicit statements are coming from. The provenance plugin provides the opposite approach. With the configured ruleset, the reasoner does forward-chaining over a specific named graph and generates all its implicit statements at query time.

2.6.3.1 Predicates

The plugin predicates give you an easy access to the graph whose implicit statements you want to generate. The process is similar to RDF reification. All plugin predicates start with `<http://www.ontotext.com/provenance/>`:

Plugin predicates	Semantics
`http://www.ontotext.com/provenance/derivedFrom`	Creates a request scope for the graph with the inference closure
`http://www.ontotext.com/provenance/subject`	Binds all subjects part of the inference closure
`http://www.ontotext.com/provenance/predicate`	Binds all predicates part of the inference closure
`http://www.ontotext.com/provenance/object`	Binds all objects part of the inference closure

2.6.3.2 Registering the plugin

The plugin is not registered by default.

1. To register it, start GraphDB with the following parameter:

```
./graphdb -Dregister-plugins=com.ontotext.trree.plugin.provenance.ProvenancePlugin
```

2. Check the startup log to validate that the plugin has started correctly.

```
[INFO ] 2016-11-18 19:47:19,134 [http-nio-7200-exec-2 c.o.t.s.i.PluginManager] Initializing plugin
↪'provenance'
```

2.6.3.3 Usage and examples

1. In the Workbench SPARQL editor, execute the following update request to add the data shown as a schema transaction:

```
PREFIX ex: <http://example.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
INSERT data {
    [] <http://www.ontotext.com/owlim/system#schemaTransaction> [] .

        ex:BusStop a rdfs:Class .
        ex:SkiResort a rdfs:Class .
        ex:WebCam a rdfs:Class .
        ex:OutdoorWebCam a rdfs:Class .
        ex:Place a rdfs:Class .

        ex:BusStop rdfs:subClassOf ex:Place .
        ex:SkiResort rdfs:subClassOf ex:Place .
        ex:WebCam rdfs:subClassOf ex:Place .
        ex:OutdoorWebCam rdfs:subClassOf ex:WebCam .

}
```

2. Execute this update request to add the data shown as a normal transaction:

```
PREFIX ex: <http://example.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

(continues on next page)

(continued from previous page)

```
INSERT data {
    GRAPH ex:g1{
        ex:webcam_g1 a ex:OutdoorWebCam .
        ex:webcam_g1 ex:containedIn ex:skiresort .
    }
     GRAPH ex:g1a{
        ex:webcam_g1a a ex:OutdoorWebCam .
        ex:webcam_g1a ex:containedIn ex:skiresort .
    }
    GRAPH ex:g2{
        ex:skiresort a ex:SkiResort .
        ex:skiresort ex:publicTransport ex:busstop .

    }
    GRAPH ex:g3{
        ex:busstop a ex:BusStop .
        ex:busstop ex:nearBy ex:skiresort .
    }
}
```

3. If we run the following query not using the plugin, it will return solutions over all statements that were inferred during the loading of the data:

```
PREFIX ex: <http://example.com/>
SELECT * {
    ?webCam a ex:WebCam .
    ?webCam ex:containedIn ?skiresort .
    ?busstop ex:nearBy ?skiresort .
}
```

   The result will have two solutions.

4. This next query showcases the newly introduced Provenance plugin predicate `pr:derive`. The computed closure is accessible in a dedicated special context whose name is provided as a subject of that predicate pattern. The patterns in the scope of that context are evaluated over the computed closure by the provenance plugin with the selected data. This allows for flexibility of use, as more than one context can be selected to supply data for processing — for example, `pr:derive (ex:g1 ex:g2)`.

```
PREFIX pr: <http://www.ontotext.com/provenance/>
PREFIX ex: <http://example.com/>
select * {
    pr:derive1 pr:derive (ex:g1 ex:g2 ex:g3)  .
    GRAPH pr:derive1
    {
        ?webCam a ex:WebCam .
        ?webCam ex:containedIn ?skiresort .
        ?busstop ex:nearBy ?skiresort .
    }
}
```

   This will return one solution since only the data from `ex:g1`, `ex:g2`, and `ex:g3` will be used. This way, the solution dependent on the data within `ex:g1a` will not be part of the result.

Note: During evaluation, the inferences and the data are kept in memory, so the plugin should be used with relatively small sets of statements placed in contexts.

2.7 Data storage

2.7.1 GraphDB persistence strategy

GraphDB stores all of its data, such as statements, indexes, and the entity pool, in files in the configured storage directory, usually called `storage`. The content and names of these files is not defined and is subject to change between versions.

There are several types of indexes available, all of which apply to all triples, whether explicit or implicit. These indexes are maintained automatically.

In general, the index structures used in GraphDB are chosen and optimized to allow for efficient:

- handling of billions of statements under reasonable RAM constraints
- query optimization
- transaction management

GraphDB maintains two main indexes on statements for use in inference and query evaluation: the predicate-object-subject (POS) index and the predicate-subject-object (PSO) index.

2.7.2 GraphDB indexing options

GraphDB includes indexing options that offer considerable advantages for specific datasets, retrieval patterns and query loads. Most of them are disabled by default but can be enabled as necessary.

Note: Unless stated otherwise, GraphDB lets you switch indexes on and off against an already populated repository. You must shut down the repository before changing the configuration. The next time the repository is started, GraphDB will create or remove the corresponding index. If the repository holds a large volume of data, switching on a new index means that the time required to rebuild that index can lead to considerable delays during initialization.

2.7.2.1 Transaction control

Transaction support is exposed via RDF4J's `RepositoryConnection` interface. The following three methods of this interface give you greater control when updates are committed to the repository:

Method	Effect
`void begin()`	Begins a transaction. Subsequent changes effected through update operations will only become permanent after `commit()` is called.
`void commit()`	Commits all updates that have been performed through this connection since the last call to `begin()`.
`void rollback()`	Rolls back all updates that have been performed through this connection since the last call to `begin()`.

GraphDB supports the so called 'read committed' transaction isolation level, which is well-known to relational database management systems — for example, pending updates are not visible to other connected users, until the complete update transaction has been committed. It guarantees that changes will not affect query evaluation before the entire transaction they are part of is successfully committed. It does not guarantee that execution of a single transaction is performed against a single state of the data in the repository. Regarding concurrency:

- Update transactions are processed internally in sequence — that is, GraphDB processes the commits one after another.

- Update transactions do not block read requests in any way. Because GraphDB is properly multi-threaded, it can evaluate hundreds of SPARQL queries in parallel while it handles update transactions on separate threads.
- Multiple update, modification, or write transactions cannot be initiated and stay open simultaneously. When a transaction is initiated and has begun to modify the underlying indexes, no other transaction must be allowed to change anything until the first one is either committed or rolled back.

Note: GraphDB performs materialization, ensuring that all statements that can be inferred from the current state of the repository are indexed and persisted (except for those compressed due to the *Optimization of owl:sameAs*). When the commit method is completed, all reasoning activities related to the changes in the data introduced by the corresponding transaction will have already been performed.

2.7.2.2 Predicate lists

Certain datasets and certain kinds of query activities — for example, queries that use wildcard patterns for predicates — benefit from another type of index called a "predicate list" such as a subject-predicate (SP) or object-predicate (OP) index.

This index maps from entities (subject or object) to their predicates. It is not switched on by default (see the *enablePredicateList configuration parameter*), because it is not always necessary. Indeed, for most datasets and query loads, the performance of GraphDB without such an index is good enough even with wildcard-predicate queries, and the overhead of maintaining this index is not justified. You should consider using this index for datasets that contain 1,000 or more different predicates.

2.7.2.3 Context index

The Context index can be used to speed up query evaluation when searching statements via their context identifier. To switch the CPSO index on or off, use the *enable-context-index configuration parameter*. The default value is `false`.

2.7.2.4 Literal index

GraphDB automatically builds a literal index that enables faster lookups of numeric and date-time object values. The index is used during query evaluation only if a query or a subquery (for example, a `UNION` clause) has a filter that is comprised of a conjunction of literal constraints using comparisons and equality (not negation or inequality), such as `FILTER(?x = 100 && ?y <= 5 && ?start > "2001-01-01"^^xsd:date)`.

Other patterns will not use the index — that is, filters will not be re-written into usable patterns.

For example, the following FILTER patterns will all make use of the literal index:

```
FILTER( ?x = 7 )
FILTER( 3 < ?x )
FILTER( ?x >= 3 && ?y <= 5 )
FILTER( ?x > "2001-01-01"^^xsd:date )
```

whereas these FILTER patterns will not:

```
FILTER( ?x > (1 + 2) )
FILTER( ?x < 3 || ?x > 5 )
FILTER( (?x + 1) < 7 )
FILTER( ! (?x < 3) )
```

The query optimizer uses statistics to decide whether to use this index. If the estimated number of matches for a filter constraint is large relative to the rest of the query — for example, a constraint with a large or one-sided range — then the index might not be used at all.

To disable this index during query evaluation, use the *enable-literal-index configuration parameter*. The default value is `true`.

Note: Because of the way the literals are stored, the index with dates far in the future and far in the past (approximately 200,000,000 years) as well as numbers beyond the range of 64-bit floating-point representation (above approximately 1e309 and below -1e309) will not work properly.

2.7.2.5 Handling of explicit and implicit statements

As already described, GraphDB applies the *inference rules at load time* in order to compute the full closure. Therefore, a repository will contain some statements that are explicitly asserted and other statements that exist through implication. In most cases, clients will not be concerned with the difference, but there are some scenarios when it is useful to work with only explicit or only implicit statements. These two groups of statements can be isolated during programmatic statement retrieval using the RDF4J API and during SPARQL query evaluation.

Retrieving statements with the RDF4J Java API

The usual technique for retrieving statements is to use the `RepositoryConnection` method:

```
RepositoryResult<Statement> getStatements(
    Resource subj,
    URI pred,
    Value obj,
    boolean includeInferred,
    Resource... contexts)
```

The method retrieves statements using a triple pattern, where any or all of the subject, predicate and object parameters can be `null` to indicate wildcards.

To retrieve explicit and implicit statements, the `includeInferred` parameter must be set to `true`. To retrieve only explicit statements, the `includeInferred` parameter must be set to `false`.

However, the RDF4J API does not provide the means to enable only the retrieval of implicit statements. In order to allow clients to do this, GraphDB allows the use of the special 'implicit' pseudo-graph with this API, which can be passed as the context parameter.

The following example shows how to retrieve only implicit statements:

```
RepositoryResult<Statement> statements =
    repositoryConnection.getStatements(
    null, null, null, true,
    SimpleValueFactory.getInstance().createIRI("http://www.ontotext.com/implicit"));

while (statements.hasNext()) {
    Statement statement = statements.next();
    // Process statement
}
statements.close();
```

The above example uses wildcards for subject, predicate and object and will therefore return all implicit statements in the repository.

SPARQL query evaluation

GraphDB also provides mechanisms to differentiate between explicit and implicit statements during query evaluation. This is achieved by associating statements with two pseudo-graphs (explicit and implicit) and using special system URIs to identify these graphs.

Note: To learn more, see *How to query explicit and implicit statements*.

2.8 Monitoring and terminating queries and update requests

You can monitor and terminate queries and update requests manually from the Workbench and automatically by configuring GraphDB to abort them after a certain *query timeout* period is reached.

2.8.1 Query monitoring and termination using the Workbench

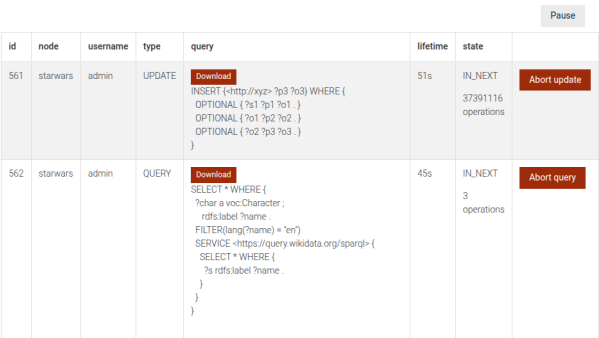
When there are running queries or update requests, their numbers are shown in the upper right next to the repositories dropdown menu.

To track and interrupt long running queries:

1. Go to *Monitor* ‣ *Queries and Updates* or click the *Running queries* status next to the repositories dropdown menu. This displays a table of any running queries or update requests.
2. Click the *Abort query* or *Abort update* button on a table row to stop the execution of one of those.

To pause the current state of the running queries, use the *Pause* button. Note that this will not stop their execution on the server.

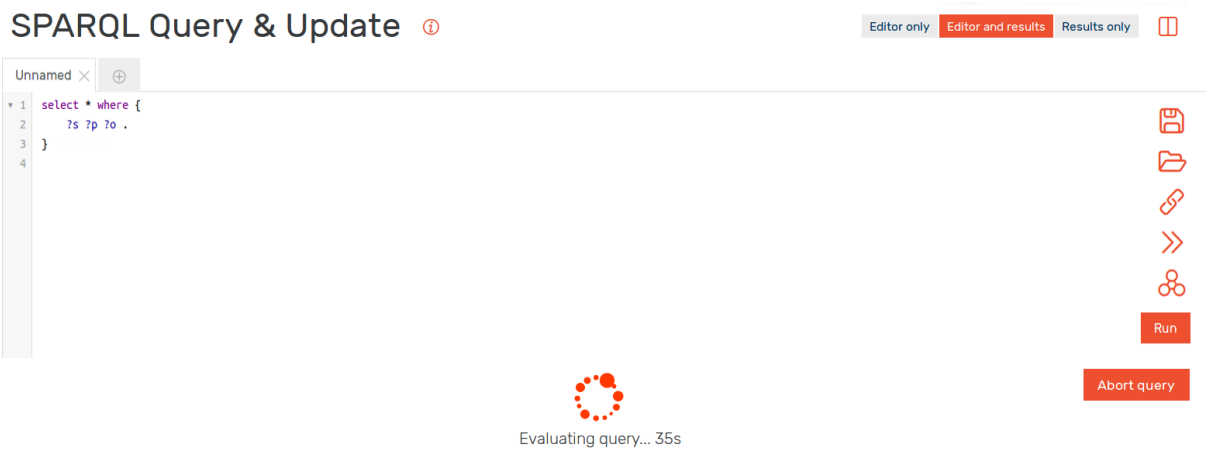
The following shows the monitor while both an update request and a query are running. (The update request and query in this example are deliberately inefficient examples so that they would display on the monitor long enough to take the screenshot. Most are displayed there only very briefly.)



The table of running queries or update request has columns with the following information about those queries:

`id`	The ID of the query
`node`	Local or remote node repository ID
`username`	The ID of the user who initiated the query
`type`	The operation type QUERY or UPDATE
`query`	The first 500 characters of the query string
`lifetime`	The time in seconds since the iterator was created
`state`	The internal state of the operation

You can also interrupt a query or update request directly from the SPARQL Editor with the *Abort query* button in the lower right:



2.8.2 Automatically prevent long running queries

You can set a global query timeout period by adding a *query-timeout* configuration parameter. All queries will stop after a number of seconds defined by this parameter, where the default value of 0 indicates no limit.

You can also control whether a terminated query returns partial results or an error using the *throw-QueryEvaluationException-on-timeout* parameter. By default, this parameter is set to `false`.

- If the parameter is set to `true`, upon reaching a timeout, GraphDB will return an error and no results.
- If the parameter is set to `false`, upon reaching a timeout, GraphDB will return partial results that were received before the timeout was reached. However, GraphDB will not indicate that the results are incomplete.

2.9 Accessing relational databases with data virtualization

2.9.1 Overview and features

GraphDB's data virtualization capabilities enable direct access using SPARQL to relational databases, eliminating the need to replicate data. Data virtualization is a useful tool when working with highly dynamic data, as well as with data sources that are too big to replicate. Moreover, in practice it is easier not to copy data and accept all limitations such as data quality, integrity, and types of supported queries of the underlying information source.

A common scenario is to maintain a declarative mapping between the relational model and RDF, where the user periodically dumps all statements and writes them to a native RDF database so it can support property paths and faster data joins.

GraphDB's relational virtualization implementation exposes a virtual SPARQL endpoint that translates the queries to SQL using declarative mapping. It achieves this through an integration with the open-source Ontop project, extended with multiple GraphDB-specific features.

GraphDB uses *Ontop version 5.3.0*. As of June 2023, Ontop supports the following database systems: PostgreSQL, MySQL, MariaDB (since 5.0.0), SQL Server, Oracle, DB2, Snowflake (since 5.0.0), Databricks (since 5.0.0), Google BigQuery (since 5.0.2), AWS Redshift (since 5.0.2), DuckDB (since 5.0.2) and AWS DynamoDB (since 5.1.0); and database federators such as Denodo, Dremio (since 4.1.0), Teiid (since 4.1.1), Apache Spark (since 4.2.0) and Trino / PrestoDB / AWS Athena (since 5.0.2).

See more about the Ontop framework in its official documentation.

2.9.2 Setup and configuration

To expose a virtual endpoint as a repository in GraphDB, you must first load the relational database into an RDBMS of your choice.

2.9.2.1 JDBC driver

Before creating a virtual repository in GraphDB, you must first install a JDBC driver for your relational database (for example, a PostgreSQL JDBC driver or a MySQL JDBC driver).

In the `lib` directory of the GraphDB distribution, create a subdirectory called `jdbc` and place the driver `.jar` file there.

Note: The driver can also be placed in the `lib` directory, but this requires a restart of GraphDB.

If you want to set a JDBC driver directory different from the `lib/jdbc` location, you can define it via the `graphdb.ontop.jdbc.path` property in the `conf/graphdb.properties` file of the GraphDB distribution.

2.9.2.2 Configuration files

In addition to the relational database and the JDBC driver, there are several other files used to create a virtual repository.

Note: You can download examples versions of those files in the *usage scenario section* below.

Required files

1. An R2RML or OBDA mapping file describing the mapping of SPARQL queries to SQL data

Optional files

1. An OWL ontology file describing the ontology of your data
2. A constraint file providing user-supplied implicit DB Constraints (e.g. unique constraints and foreign keys) to Ontop to generate efficient SQL
3. An Ontop lenses file that specifies integrity constraints on views defined at the Ontop level
4. A DB metadata file that provides extracted database information such as DB connection metadata, DB schema and integrity constraints (for instance primary keys)

Automatically generated files

There are also several files that are automatically generated when you create a virtual repository through the Workbench. Unless you are creating such a repository using the `curl` command as described in *Creating a virtual repository using curl* below, you do not need to worry about those files.

1. A properties file for the configuration of the JDBC driver parameters of the following type:

```
jdbc.url=<database-jdbc-driver-connection-string>
jdbc.driver=<database-jdbc-driver-class>
jdbc.user=<your-database-username>
jdbc.password=<your-database-password>
```

2. A repository config file of the following type that references the aforementioned properties, R2RML or OBDA mapping, ontology, constraints, lenses, and DB metadata files:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rep: <http://www.openrdf.org/config/repository#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#example-virtual> a rep:Repository;
  rep:repositoryID "example-virtual";
  rep:repositoryImpl [
      <http://inf.unibz.it/krdb/obda/quest#propertiesFile> "example.properties";
      <http://inf.unibz.it/krdb/obda/quest#obdaFile> "example-mapping.obda";
      <http://inf.unibz.it/krdb/obda/quest#owlFile> "example-ontology.ttl";
      <http://inf.unibz.it/krdb/obda/quest#constraintFile> "example-constraints.lst";
      <http://inf.unibz.it/krdb/obda/quest#lensesFile> "example-lenses.json";
      <http://inf.unibz.it/krdb/obda/quest#dbMetadataFile> "example-db-metadata.json";
      rep:repositoryType "graphdb:OntopRepository"
    ];
  rdfs:label "Ontop virtual store with OBDA" .
```

Ontop configuration properties and query logging

Ontop configuration properties can be configured through the *GraphDB config file* using properties whose names fit the pattern `graphdb.ontop.<property>`. The `graphdb.` part will be stripped and the resulting `ontop.<property>` property will be passed to Ontop.

For example, to configure Ontop query logging, set the following properties:

```
# Enables query logging
graphdb.ontop.queryLogging = true

# Includes the SPARQL query string into the query log (true by default)
graphdb.ontop.queryLogging.includeSparqlQuery = true

# Includes the reformulated (SQL) query into the query log (false by default)
graphdb.ontop.queryLogging.includeReformulatedQuery = true
```

The Ontop log will be written the same way as any other *GraphDB log messages*.

2.9.3 Creating a virtual repository from the Workbench

To create a new virtual repository, *create a new repository* and select Ontop Virtual SPARQL as the repository type.

When creating an Ontop repository in GraphDB, the default setting is *Generic JDBC Driver* where you need to specify the Driver class yourself. You also have the option to create an Ontop repository by using the predefined settings for one of several commonly used drivers.

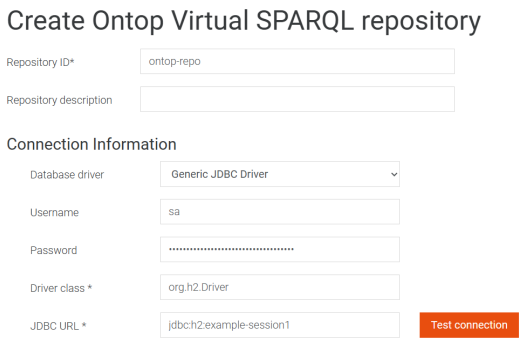
Warning: Once you have created an Ontop repository, its type cannot be changed.

2.9.3.1 With a generic JDBC driver

Connection information

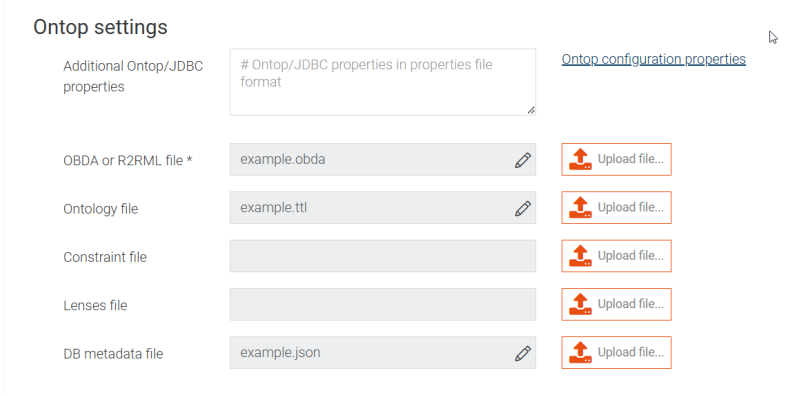
1. After you create a new Ontop repository, leave the default setting as *Generic JDBC Driver*.
2. Enter your *Username* and *Password*, if applicable.
3. Define your *Driver class*.
4. Enter your *JDBC URL*.

Once you have filled in this information, you can test the connection to your SQL database with the *Test connection* button on the right.



Ontop settings

1. Upload an R2RML or OBDA file to fill in the *OBDA or R2ML file* field. The *Ontology*, Constraint, Lenses, and *DB metadata* files are optional.
2. To include optional properties, type them in the field for *Additional Ontop/JDBC properties*, using the properties file format.
3. You can test the connection to your SQL database with the button on the right.
4. Click *Create*.



2.9.3.2 With one of the other supported database drivers

For ease of use, GraphDB also supports drivers for eight other commonly used database managers integrated into the Ontop framework: MySQL, PostgreSQL, Oracle, MS SQL Server, DB2, Dremio, Databricks, and Snowflake. Those options come with predefined *Driver class* properties. When working with one of those options, the *URL* property values are generated by GraphDB.

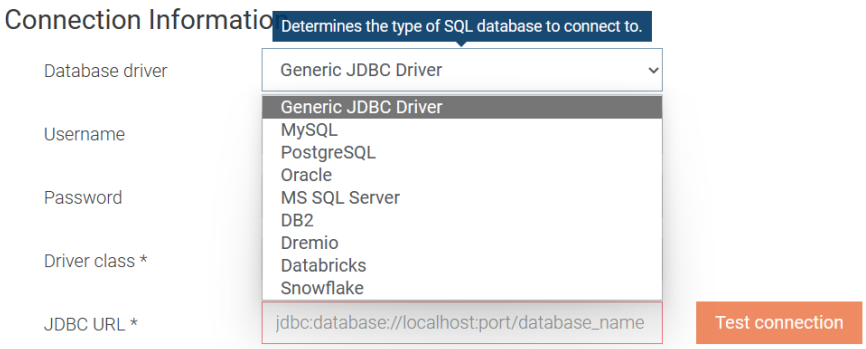
To use one of these database drivers:

Connection information

1. Select the type of SQL database you want to use from the drop-down menu.
2. Fill in the required fields for each driver such as *Hostname* and *Database name*.
3. Optionally, fill in the *Port*, *Username* and *Password* properties.

Note: After you select the type of SQL database, clicking the *Download JDBC driver* link on the right of the *Driver class* field will perform a web search for the corresponding driver. For more information on how to set up your JDBC driver, read the *JDBC driver* section above.

Note: Although you cannot modify the existing provided template, you can still add to the *JDBC URL* field value after it is automatically generated. This is required when working with complex data sources such as Databricks that expect additional properties in order to establish the connection.



Tip: This form's *Ontop settings* are covered in the *Ontop settings* section above.

2.9.3.3 Creating a virtual repository using curl

To create a virtual repository via the API, you will need an R2RML or OBDA mapping file, as described above. You will also need the following two files to provide information that would have been provided through a form when creating a virtual repository with Workbench:

- `example-repo-config.ttl`: an RDF4J repository configuration file
- `example.properties`: a JDBC properties file

The following additional files are optional:

- `example-ontology.ttl`: the Ontop ontology file
- `example-constraints.lst`: the Ontop constraint file
- `example-lenses.json`: the Ontop lenses file

- `example-db-metadata.json`: the Ontop DB metadata file

You can see a description of each file in the *Configuration files* section above.

The following shows an example `curl` command to create a virtual repository with the API with the the required and optional files:

```
curl -X POST http://localhost:7200/rest/repositories -H "Content-Type: multipart/form-data"\
    -F "config=@example-repo-config.ttl"\
    -F "propertiesFile=@example.properties"\
    -F "obdaFile=@example-mapping.obda"\
    -F "owlFile=@example-ontology.ttl"\
    -F "constraintFile=@example-constraints.lst"\
    -F "lensesFile=@example-lenses.json"\
    -F "dbMetadataFile=@example-db-metadata.json"
```

You will see the newly created repository under *Setup* ▸ *Repositories* in the GraphDB Workbench.

2.9.4 Mapping language

The underlying Ontop engine supports two mapping languages. The first one is the official W3C RDB2RDF mapping language known as R2RML, which provides excellent interoperability between the various tools. The second one is the native Ontop mapping language known as OBDA, which is easier to learn, supports more concise mapping specifications, and supports an automatic bidirectional transformation to R2RML.

2.9.5 SPARQL endpoint

For a comprehensive list of supported SPARQL features, please see Ontop's documentation on standard compliance.

You can see examples of SPARQL queries supported by GraphDB virtual repositories in the *SPARQL example queries* section.

2.9.6 Query federation

GraphDB also supports querying the virtual read-only repositories using the highly efficient *Internal SPARQL federation*.

Its usage is the same as with the internal federation of regular repositories. Instead of providing a URL to a remote repository, you need to provide a special URL of the form `repository:NNN`, where `NNN` is the ID of the virtual repository you want to access.

You can see an example usage scenario of query federation in the *Query federation example* section.

2.9.7 Usage scenario

Let's use a relational database containing university data.

Note:

You can download the required files for this usage scenario below:

1. `The university SQL data` (import it into an SQL database of your choice).
2. `OBDA mapping file`.
3. `Ontology file`.
4. `DB metadata file`.

5. `Config file for the repository` (necessary when creating a virtual repository using curl).
6. `JDBC Properties file` (necessary when creating a virtual repository using curl).

It has tables describing students, academic staff, courses and two relational schemas (*uni1* and *uni2*) with many-to-many links between academic staff ‣ courses and students ‣ courses. The descriptions below are for the *uni1* tables.

Table 1: *uni1.student*

s_id	first_name	last_name
1	Mary	Smith
2	John	Doe

The table above contains the local ID and the first and last names of the students. The column *s_id* is a primary key.

Table 2: *uni1.academic*

a_id	first_name	last_name	position
1	Anna	Chambers	1
2	Edward	May	9
2	Rachel	Ward	8

Similarly, this table contains the local ID and the first and last names of the academic staff, but also information about their position. The column *a_id* is a primary key.

The column *position* is populated with magic numbers:

- 1 ‣ Full Professor
- 2 ‣ Associate Professor
- 3 ‣ Assistant Professor
- 8 ‣ External Teacher
- 9 ‣ PostDoc

Table 3: *uni1.course*

c_id	title
1234	Linear Algebra

This table contains the local IDs and titles of the courses. The column *c_id* is a primary key.

Table 4: *uni1.teaching*

c_id	a_id
1234	1
1234	2

This table contains the n-n relation between courses and teachers. There is no primary key, but two foreign keys to the tables *uni1.course* and *uni1.academic*.

Table 5: *uni1.course-registration*

c_id	s_id
1234	1
1234	2

This table contains the n-n relation between courses and students. There is no primary key, but two foreign keys to the tables *uni1.course* and *uni1.student*.

2.9.7.1 SPARQL example queries

Below are some examples of the SPARQL queries that are supported in a GraphDB virtual repository.

1. Return the IDs of all persons that are faculty members:

```
PREFIX voc: <http://example.org/voc#>

SELECT ?p {
    ?p a voc:FacultyMember .
}
```

2. Return the IDs of all full Professors together with their first and last names:

```
PREFIX voc: <http://example.org/voc#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?prof ?lastName ?firstName {
    ?prof a voc:FullProfessor ;
          foaf:firstName ?firstName ;
          foaf:lastName ?lastName .
}
```

3. Return all Associate Professors, Assistant Professors, and Full Professors with their last names and first name if available, and the title of the course they are teaching:

```
PREFIX voc: <http://example.org/voc#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?title ?fName ?lName {
    ?teacher rdf:type voc:Professor .
    ?teacher voc:teaches ?course .
    ?teacher foaf:lastName ?lName .

    ?course voc:title ?title .
    OPTIONAL {
        ?teacher foaf:firstName ?fName .
    }
}
```

2.9.7.2 Query federation example

Let's see how querying using *Internal SPARQL federation* works with our university database example.

1. Create a new, empty RDF repository called `university-rdf`.
2. From the `ontop_repo` virtual repository with university data, insert some data in the new, empty `university-rdf` repository: teachers with first name and last name values that give courses that are **not** held at university2:

```
PREFIX voc: <http://example.org/voc#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT {
    ?person a voc:UniversityTeacher;
            voc:firstName ?firstName;
            voc:lastName ?lastName .
} WHERE {
    SERVICE <repository:my_repo> {
        SELECT DISTINCT ?person ?firstName ?lastName {
            ?person foaf:firstName ?firstName ;
```

(continues on next page)

(continued from previous page)

```
                    foaf:lastName ?lastName ;
                    voc:teaches [ voc:isGivenAt ?institution ]
            FILTER(?institution != <http://example.org/voc#uni2/university>)
        }
    }
}
```

3. To observe the results, again in the `university-rdf` repository, execute the following query that will return the teachers that were inserted with their first and last name:

```
PREFIX voc: <http://example.org/voc#>

SELECT * {
    ?teacherId a voc:UniversityTeacher;
        voc:firstName ?firstName;
        voc:lastName ?lastName;
}
```

Result:

Table | Raw Response | Pivot Table | Google Chart | Download as

Filter query results — Showing results from 1 to 17 of 17. Query took 0.1s, minutes ago.

	teacherId	firstName	lastName
1	http://example.org/voc#uni1/academic/8	"Alois"	"Jayant"
2	http://example.org/voc#uni1/academic/3	"Rachel"	"Ward"
3	http://example.org/voc#uni1/academic/14	"Karine"	"Attilio"
4	http://example.org/voc#uni1/academic/1	"Anna"	"Chambers"
5	http://example.org/voc#uni1/academic/2	"Edward"	"May"

4. Then:

- get the teachers from the virtual repository that teach courses in an institution that is **not** university2
- merge the result of that with the RDF repository by getting the `firstName` and `lastName` of those teachers

The IDs of the teachers are the common property for both repositories, which makes the selection possible. For the purposes of this demonstration, this query filters them by `firstName` that contains the letter "a".

```
PREFIX voc: <http://example.org/voc#>

SELECT * {
    SERVICE <repository:ontop_repo> {
        ?teacherId voc:teaches [ voc:isGivenAt ?institution] .
        FILTER (?institution != "http://example.org/voc#uni2/university")
    }
    ?teacherId voc:firstName ?firstName;
               voc:lastName ?lastName
    FILTER (regex(?firstName, "a"))
}
```

Result:

Table | Raw Response | Pivot Table | Google Chart | Download as

Filter query results — Showing results from 1 to 15 of 15. Query took 0.3s, minutes ago.

	teacherId	institution	firstName	lastName
1	http://example.org/voc#uni1/academic/1	http://example.org/voc#uni1/university	"Anna"	"Chambers"
2	http://example.org/voc#uni1/academic/13	http://example.org/voc#uni1/university	"Gerard"	"Cosimo"
3	http://example.org/voc#uni1/academic/2	http://example.org/voc#uni1/university	"Edward"	"May"
4	http://example.org/voc#uni1/academic/14	http://example.org/voc#uni1/university	"Karine"	"Attilio"
5	http://example.org/voc#uni1/academic/1	http://example.org/voc#uni1/university	"Anna"	"Chambers"
6	http://example.org/voc#uni1/academic/3	http://example.org/voc#uni1/university	"Rachel"	"Ward"

2.9.8 Limitations

Data virtualization comes with certain limitations due to the distributed nature of the data. In this sense, it works best for information that requires little or no integration. For instance, if databases X and Y have two instances of the person John Smith, which do not share a unique key or other exact match attributes like "John Smith" and "John E. Smith", it will be quite inefficient to match the records at runtime.

When using a virtual repository, keep the following in mind:

- Virtual repositories are read-only, meaning that write operations cannot be executed in them
- sameAs is disabled
- Running an Alpine Linux Based Docker GraphDB with a DuckDB Driver will result in a Runtime Exception
- GraphDB explain plan is not available

One potential drawback is also the type of supported queries. If the underlying storage has no indexes, it will be slow to answer queries such as "tell me how resource X connects to resource Y".

The number of stacked data sources also significantly affects the efficiency of data retrieval.

Lastly, it is not possible to efficiently use auto-suggest or auto-complete indexes nor perform graph traversals or inferencing.

2.10 Federation from multiple SPARQL endpoints with FedX

2.10.1 Overview

In addition to the standard SPARQL 1.1 Federation to other SPARQL endpoints and the *internal SPARQL federation* to other repositories in the same database instance, GraphDB also supports FedX — the federation engine of the RDF4J framework, a data partitioning technology that provides transparent federation of multiple SPARQL endpoints under a single virtual endpoint.

With the growing need for scalability of RDF technology and sophisticated optimization techniques for querying linked data, it is a useful framework that allows efficient SPARQL query processing on heterogeneous, virtually integrated linked data sources. With it, explicitly addressing specific endpoints using SERVICE clauses is no longer necessary — instead, FedX offers novel join processing and grouping techniques to minimize the number of remote requests by automatically selecting the relevant sources, sending statement patterns to these sources for evaluation and joining the individual results. It extends the RDF4J framework with a federation layer and is incorporated in it as a *SAIL* (Storage and Inference Layer).

Note: Please keep in mind that the GraphDB FedX federation is currently an experimental feature.

2.10.2 Features

GraphDB supports the following FedX features:

- Virtual joins of distributed knowledge graphs: Following the idea of the Linked Open Data Initiative for connecting distributed RDF data, FedX federation combines distributed data sources with the goal of virtual interaction. This means that data from multiple heterogeneous sources can be queried transparently as if they were in the same database.
- Federation of sharded knowledge graphs: A virtual knowledge graph can consist of various knowledge subgraphs distributed in separate endpoints and can be virtually integrated using FedX Join. FedX is optimized for such scenarios where each graph has a different schema — that is, the graph is separated into exclusive groups.
- Easy integration as a GraphDB repository

- Transparent access to data sources through federation
- Streamlined query processing in federated environments

2.10.3 Usage scenarios

The following sections demonstrate how semantic technology that uses FedX federation lowers the cost of search and analysis of data sources by implementing a two-step integration process:

1. Mapping any dataset to an ontology.
2. Using GraphDB to access the data.

   With this integration methodology, the cost of extending the number of supported sources remains linear, unlike the classic data warehousing approaches.

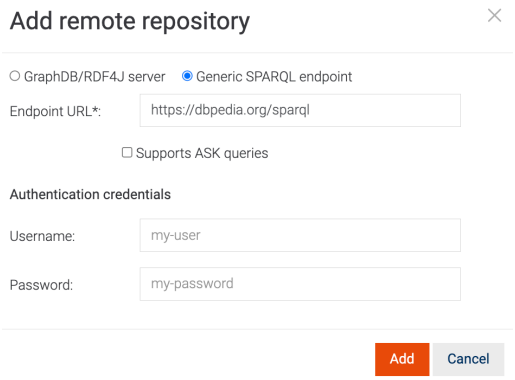
2.10.3.1 Linked data cloud exploration

For our first use case we will create a unifying repository where we can query data from multiple linked data sources regardless of their location, such as DBpedia and Wikidata. In such cases, there is often a significant overlap between the schemas such as predicates or types that are frequently repeated across the different sources.

Note: Keep in mind that blank nodes are not supported between FedX members.

Before we start exploring, let's first create a federation between the DBpedia and Wikidata endpoints.

1. Create a FedX repository via *Setup* ‣ *Repositories* ‣ *Create new repository* ‣ *FedX Virtual SPARQL*.
2. On the *Create FedX Virtual SPARQL repository* screen, click *Add remote repository*.
3. On the *Add remote repository* dialog box select *Generic SPARQL endpoint*.
4. For the DBpedia *Endpoint URL*, enter `https://dbpedia.org/sparql`.
5. Unselect the *Supports ASK queries* option, as this differs from endpoint to endpoint and click the dialog box's *Add* button. (You can ignore the *Authentication credentials*, which are unnecessary for DBpedia.)



6. Click *Add remote repository* and repeat the same steps for the Wikidata endpoint URL: `https://query.wikidata.org/sparql`.
7. Click *Create* in the lower right of the *Create FedX Virtual SPARQL repository* form to save the new repository and connect to it on the *Repositories* screen.

Now, let's perform some queries against the federated repository.

Scenario 1: Querying one of the endpoints using predicates and nodes specific to it

The following query is run against the Wikidata endpoint and will return all instances of **"house cat"**.

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT * WHERE
{
    ?item wdt:P31 wd:Q146.
    ?item rdfs:label ?label.
    FILTER (LANG(?label) = 'en')
}
```

Here, we have used two Wikidata predicates: `wdt:P31` that stands for "instance of" and `wd:Q146` that stands for "house cat".

The following shows the first few results for this query:

	item	label
1	http://www.wikidata.org/entity/Q378619	"CC"@en
2	http://www.wikidata.org/entity/Q893453	"Unsinkable Sam"@en
3	http://www.wikidata.org/entity/Q1050083	"Catmando"@en
4	http://www.wikidata.org/entity/Q1201902	"Tama"@en
5	http://www.wikidata.org/entity/Q1637234	"Humphrey"@en
6	http://www.wikidata.org/entity/Q1413628	"Nora"@en
7	http://www.wikidata.org/entity/Q1622688	"Hodge"@en
8	http://www.wikidata.org/entity/Q677525	"Orangey"@en
9	http://www.wikidata.org/entity/Q851190	"Mrs. Chippy"@en
10	http://www.wikidata.org/entity/Q498787	"Muezza"@en
11	http://www.wikidata.org/entity/Q1207136	"Dewey Readmore Books"@en
12	http://www.wikidata.org/entity/Q1371145	"Socks"@en
13	http://www.wikidata.org/entity/Q1386318	"F. D. C. Willard"@en
14	http://www.wikidata.org/entity/Q1750907	"Towser"@en
15	http://www.wikidata.org/entity/Q1759652	"Wilberforce"@en

Scenario 2: Querying both endpoints

With a CONSTRUCT query, we can get data about **a specific cat** from both endpoints — CC ("CopyCat" or "Carbon Copy"), the first cloned pet.

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT {
    ?s ?p ?o
} WHERE
{
    {
        BIND(<http://www.wikidata.org/entity/Q378619> as ?s)
        ?s ?p ?o.
    } UNION {
        BIND(<http://www.wikidata.org/entity/Q378619> as ?o)
        ?s ?p ?o.
    }
}
```

	subject	predicate	object
1	http://www.wikidata.org/entity/Q378619	http://schema.org/version	"1316780076"^^xsd:integer
2	http://www.wikidata.org/entity/Q378619	http://schema.org/dateModified	"2020-12-02T13:23:00Z"^^xsd:dateTime
3	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"kazh"@br
4	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"ネコ"@ja
5	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"naklonovaná kočka"@cs
6	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"geklonte Katze"@de
7	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"cloned cat"@en
8	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"ensimmäinen kloonattu lemmikkieläin"@fi
9	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"chat cloné"@fr
10	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"gekloonde kat"@nl
11	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"gato clonado"@pt-br
12	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"pisică clonată"@ro
13	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"klonad katt"@sv
14	http://www.wikidata.org/entity/Q378619	http://schema.org/description	"klonlanmış kedi"@tr
15	http://www.wikidata.org/entity/Q378619	http://www.w3.org/2000/01/rdf-schema#label	"CC"@cs

Scenario 3: Querying both endpoints for a specific resource

Let's explore both DBpedia and Wikidata for **products** made by the company **Amazon**.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

CONSTRUCT{
    ?dbpCompany ?p ?o .
    ?wdCompany ?p1 ?o .
    ?dbpCompany owl:sameAs ?wdCompany .
} WHERE {
    BIND( dbr:Amazon_\(company\) as ?dbpCompany)
    {
        # Get all products from DBpedia
        ?dbpCompany dbo:product ?o .
        ?dbpCompany ?p ?o .
    } UNION {
        # Get all products from Wikidata
        ?dbpCompany owl:sameAs ?wdCompany .
         ?o wdt:P176 ?wdCompany .
        ?o ?p1 ?wdCompany .
    }
}
```

	subject	predicate	object
1	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/wikiPageWikiLink	http://dbpedia.org/resource/Fire_OS
2	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/property/products	http://dbpedia.org/resource/Fire_OS
3	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/product	http://dbpedia.org/resource/Fire_OS
4	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/wikiPageWikiLink	http://dbpedia.org/resource/Amazon_Echo
5	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/wikiPageWikiLink	http://dbpedia.org/resource/Amazon_Kindle
6	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/wikiPageWikiLink	http://dbpedia.org/resource/Amazon_Fire_TV
7	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/property/products	http://dbpedia.org/resource/Amazon_Echo
8	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/property/products	http://dbpedia.org/resource/Amazon_Kindle
9	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/property/products	http://dbpedia.org/resource/Amazon_Fire_TV
10	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/product	http://dbpedia.org/resource/Amazon_Echo
11	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/product	http://dbpedia.org/resource/Amazon_Kindle
12	http://dbpedia.org/resource/Amazon_(company)	http://dbpedia.org/ontology/product	http://dbpedia.org/resource/Amazon_Fire_TV
13	http://www.wikidata.org/entity/Q3884	http://www.wikidata.org/prop/direct/P176	http://www.wikidata.org/entity/Q143298
14	http://dbpedia.org/resource/Amazon_(company)	http://www.w3.org/2002/07/owl#sameAs	http://www.wikidata.org/entity/Q3884
15	http://www.wikidata.org/entity/Q3884	http://www.wikidata.org/prop/direct/P178	http://www.wikidata.org/entity/Q143298

Scenario 4: Creating an advanced graph configuration for a query

As we saw in the previous example, we can explore a specific resource from both endpoints. Now, let's see how to create an advanced graph configuration for a query, with which we can explore any resource that we input.

With the following steps, create a graph configuration query for **all companies** and **all products** in both datasets:

1. Go to *Explore* ▸ *Visual graph*.
2. From *Advanced graph configurations*, select *Create graph config*.
3. Enter a name for your graph.
4. On the first tab at the bottom of the *Create visual graph config* form, the default Starting point is, the default *Starting point* is *Start with a search box*. Next to that, select the *Graph expansion* tab and enter the following query:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

CONSTRUCT{
    ?node ?p1 ?o .
    ?s ?p ?o .
    ?node owl:sameAs ?s .
} WHERE {
    {
        ?node dbo:product ?o .
        ?node ?p1 ?o .
    } UNION {
        ?node owl:sameAs ?s .
        ?o wdt:P176 ?s .
        ?o ?p ?s .
    }
}
```

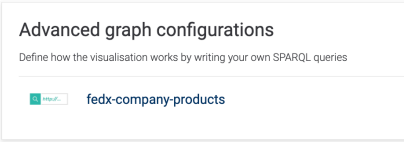
   The query connects the two databases using the `owl:sameAs` predicate. The DBpedia property `dbo:product` corresponds to the `wdt:P176` property in Wikidata.

5. Since Wikidata shows information in a less readable way, we can have our configuration clear it up a little by pulling the node labels. To do so, go to the *Node basics* tab and enter the query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?label {
    {
        ?node rdfs:label | skos:prefLabel ?label.
        FILTER (lang(?label) = 'en')
    }
}
```

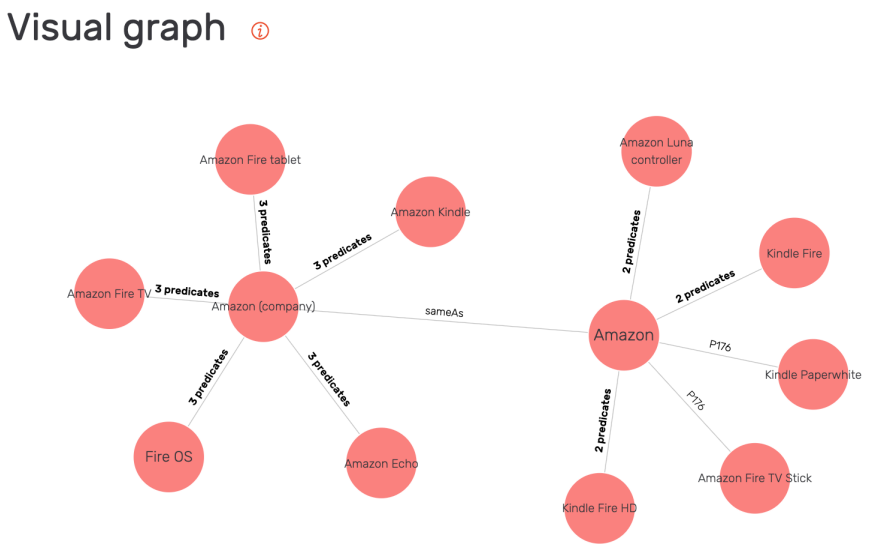
   Note: The `?node` variable is required and will be replaced with the IRI of the node that we are exploring.

6. Click *Save* in the lower right of the form. You will be taken back to the *Visual graph* screen where the newly created graph configuration is now visible with whatever name you gave it.

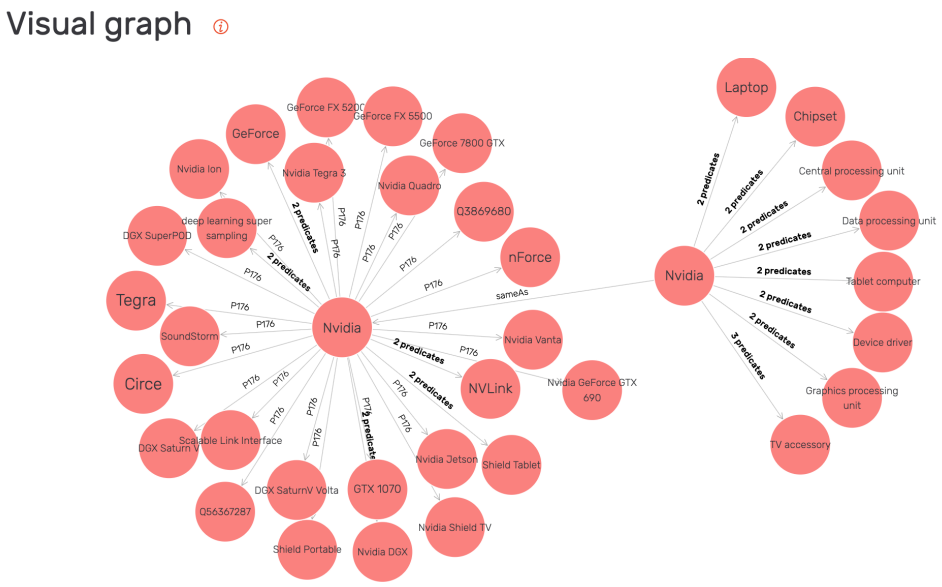


7. Now let's explore the information about the nodes as visual graphs mapped in both data sources. Click on the name of the graph configuration that you created, and in the search field that opens, enter the DBpedia resource `http://dbpedia.org/resource/Amazon_(company)` and click *Show*.

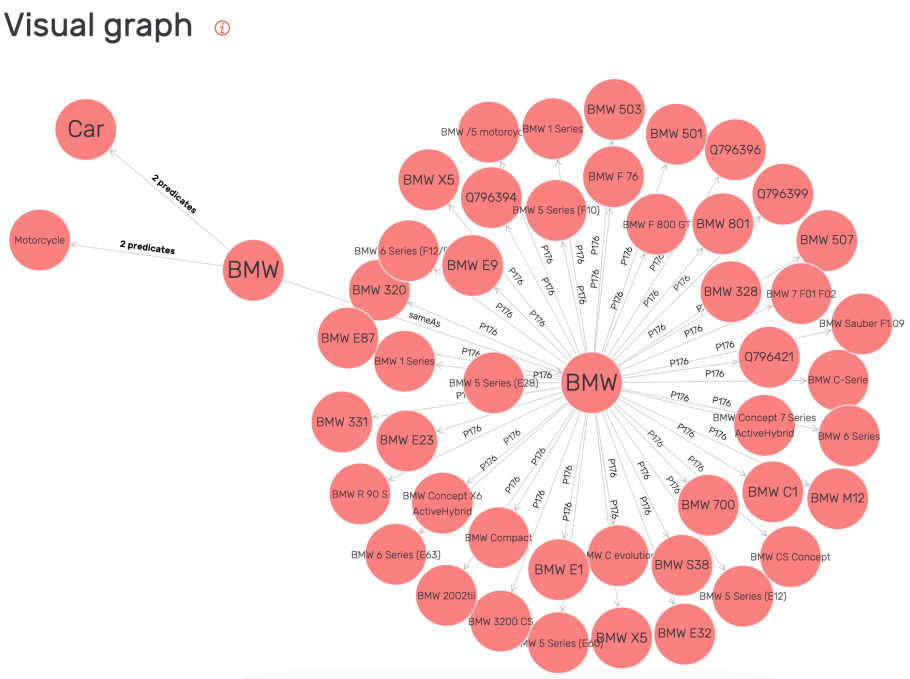
   On the left are the DBpedia resources related to Amazon, and on the right, the Wikidata ones.



We can do the same for company `http://dbpedia.org/resource/Nvidia`:



And for `http://dbpedia.org/resource/BMW`:



Note: Some SPARQL endpoints with implementation other than GraphDB may enforce limitations that could result in some features of the GraphDB FedX repository not working as expected. One such example is the class hierarchy that may send big queries and not work with `https://dbpedia.org/sparql`, which has a query length limit.

2.10.3.2 Virtual knowledge graph over local native and RDBMS-based repositories

The second type of scenario demonstrates how to create a federated repository over two local repositories — a local native one and an *Ontop* one that provides an RDF view of data from relational tables. Let's divide a dataset between them and then explore the relationships.

We will use segments of two public datasets:

- The **native GraphDB repository** uses data from the `acquisitions.csv`, `ipos.csv`, and `objects.csv` files of the Startup investments dataset, a Crunchbase snapshot data source that contains metadata about companies and investors. It tracks the evolution of startups into multi-billion corporations. The data has been converted to RDF using Ontotext Refine.
  - The `acquisitions` file contains M&A deals between companies, listing all buyers and acquired companies and the date of each deal.
  - The `objects` file contains details about the companies, such as ID, name, and country.
  - The `ipo` file contains data about company IPOs.
- The **Ontop repository** uses the `prices.csv` file of the NYSE dataset, a data source listing the opening and closing stock price and traded volumes on the New York Stock Exchange. The file lists stock symbols and opening and closing stock market prices for particular dates. Most data span from 2010 to the end 2016. For companies new on the stock market the date range is shorter.

1. To set up the **native** GraphDB repository:
   a. Create *a new repository*.
   b. Download the `ipo.nq`, `acquisitions.ttl` and `objects.ttl` files.
   c. Load them into the repository via *Import* ▸ *User data* ▸ *Upload RDF files*.
2. To set up the **Ontop** repository:

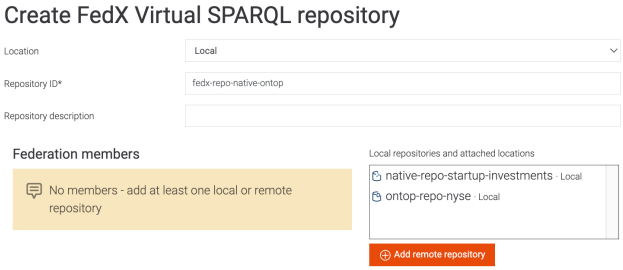
a. Download the `prices-mapping.obda` file.

b. Create an *Ontop repository* using the OBDA file.

3. To create the **FedX** repository where these two will be federated:

a. Create it via *Setup* ▸ *Repositories* ▸ *Create new repository* ▸ *FedX Virtual SPARQL*.

b. In the configuration screen, include the two repositories that we created as members by clicking on them.



c. After it has been created, connect to it.

Now that we have created the federation, let's see how we can explore the two data sources with it.

Scenario 1: List European companies acquired by US companies

The following query is run against the Crunchbase data source and returns **all companies registered in European countries** that have been **acquired by US companies**.

```
PREFIX cb: <https://crunchbase.com/resource/cb/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?sellingCompanyName ?buyerCompanyName
WHERE {
    ?id cb:buyer ?buyerCompany .
    ?id cb:target ?acquiredCompany .
    ?buyerCompany cb:country "USA" .
    ?acquiredCompany cb:country ?country .
    FILTER (?country IN ("AUT", "BEL", "BGR", "HRV", "CYP", "CZE", "DNK", "EST", "FIN", "FRA", "DEU",
↪"GBR", "GRC", "HUN", "IRL", "ITA", "LVA", "LTU", "LUX", "MLT", "NLD", "POL", "PRT", "ROU", "SVK", "SVN
↪", "ESP", "SWE"))
    ?acquiredCompany rdfs:label ?sellingCompanyName .
    ?buyerCompany rdfs:label ?buyerCompanyName .
}
```

The first two triple patterns match on the acquiring and the acquired company. The "USA" literal specifies that the buyer company is based there. The target company must be European. Each company's country is represented by a country code. To get only the European companies that have been acquired, a filter checks whether a given country's code is among the listed ones.

The first 15 returned results look like this:

	sellingCompanyName	buyerCompanyName
1	"e-dialog"	"GSI Commerce"
2	"Fatfoogoo"	"Digital River"
3	"video2brain"	"lynda.com"
4	"iPEK Spezial TV"	"IDEX Corp"
5	"SolveDirect Service Management"	"Cisco"
6	"SensorDynamics"	"Maxim Integrated Products"
7	"f-wz"	"Sabre Airline Solutions"
8	"Shopigram"	"Undisclosed Company"
9	"Uteron Pharma"	"Watson Pharmaceuticals"
10	"2dehands.be"	"eBay"
11	"2ememain.be"	"eBay"
12	"Q-Layer"	"Sun Microsystems"
13	"Mollom"	"Acquia"
14	"Constructiewerkhuizen G. Verbruggen"	"Nordson"
15	"ShareYourCart"	"AddShoppers"

Scenario 2: List European companies acquired by US companies where the stock market price of the buyer company has increased on the date of the M&A deal

This query is run against the Crunchbase and the NYSE datasets and is similar to the one above, but with one additional condition — that on the day of the deal, the **stock price of the buying company has increased**. This means that when the stock market closed, that price was higher than when the market opened. Since the M&A deals data are in the Crunchbase dataset and the stock prices data in the NYSE dataset, we will join them on the `stockSymbol` field, which is present in both datasets, and the IPO value of the buyer company.

The query also confirms that the date of the M&A deal (from Crunchbase) is the same as the date for which we retrieve the opening and closing stock prices (from NYSE). In the SELECT clause, we include only the names of the buyer and seller companies. The opening and closing prices are chosen for a particular date and stock symbol.

```
PREFIX ny: <https://www.kaggle.com/dgawlik/nyse/>
PREFIX cb: <https://crunchbase.com/resource/cb/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?sellingCompanyName ?buyerCompanyName
WHERE {
    ?id cb:buyer ?buyerCompany .
    ?id cb:target ?acquiredCompany .
    ?acquiredCompany cb:country ?country .
    ?buyerCompany cb:country "USA" .
    ?buyerCompany cb:ipo ?buyerCompanyIpo .
    ?buyerCompanyIpo cb:stockSymbol ?stockSymbol .
    ?id cb:acquiredAt ?date .
    ?dayTrade ny:date ?date .
    ?dayTrade ny:ticker ?stockSymbol .
    ?dayTrade ny:close ?close .
    ?dayTrade ny:open ?open .
    FILTER (?country IN ("AUT", "BEL", "BGR", "HRV", "CYP", "CZE", "DNK", "EST", "FIN", "FRA", "DEU",
↪"GBR", "GRC", "HUN", "IRL", "ITA", "LVA", "LTU", "LUX", "MLT", "NLD", "POL", "PRT", "ROU", "SVK", "SVN
↪", "ESP", "SWE") && (?close > ?open))
    ?acquiredCompany rdfs:label ?sellingCompanyName .
    ?buyerCompany rdfs:label ?buyerCompanyName .
}
```

The first 15 returned results look like this:

Filter query results

Showing results from 1 to 275 of 275. Query took 45s, moments ago.

	sellingCompanyName	buyerCompanyName
1	"OptiMedica"	"Abbott Labs"
2	"IDEV Technologies"	"Abbott Labs"
3	"Facet Biotech"	"Abbott Labs"
4	"Livio Radio"	"Ford"
5	"Aragon Pharmaceuticals"	"Johnson & Johnson"
6	"Calibra Medical"	"Johnson & Johnson"
7	"MobileAccess Networks"	"Corning"
8	"YaData"	"Microsoft"
9	"StorSimple"	"Microsoft"
10	"PhoneFactor"	"Microsoft"
11	"Skype"	"Microsoft"
12	"MetricsHub"	"Microsoft"
13	"Apiphany"	"Microsoft"
14	"AVIcode"	"Microsoft"
15	"RADIL"	"IDEXX Laboratories"

2.10.3.3 FedX with enabled GraphDB security

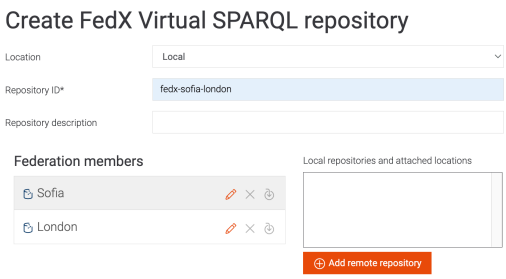
When creating a FedX repository with local members, you can specify whether the FedX repository should respect the security rights of the member repositories.

Configuring security for local members

1. First, we will create two repositories, “Sofia” and “London”, in which we will insert some statements from factforge.net:
   a. Create a repository called Sofia.
   b. Go to the FactForge SPARQL editor and execute:

```
CONSTRUCT WHERE {
    ?s ?p <http://dbpedia.org/resource/Sofia> .
} LIMIT 20
```

   c. Download the results in Turtle format.
   d. Import the file in the Sofia repository via *Import* ▸ *User data* ▸ *Upload RDF files*.
   e. Repeat the same steps for the “London” repository with FactForge data about London.
2. Create a FedX repository with the “Sofia” and “London” repositories as members.



The icons next to the name of each member are for:

- editing the repository’s access rights
- removing the repository as a FedX member (this will move it back in the repository list)
- setting the repository as writable; note that only one member repository can be writable

3. In it, execute the SPARQL editor default query that returns all statements in the repository:

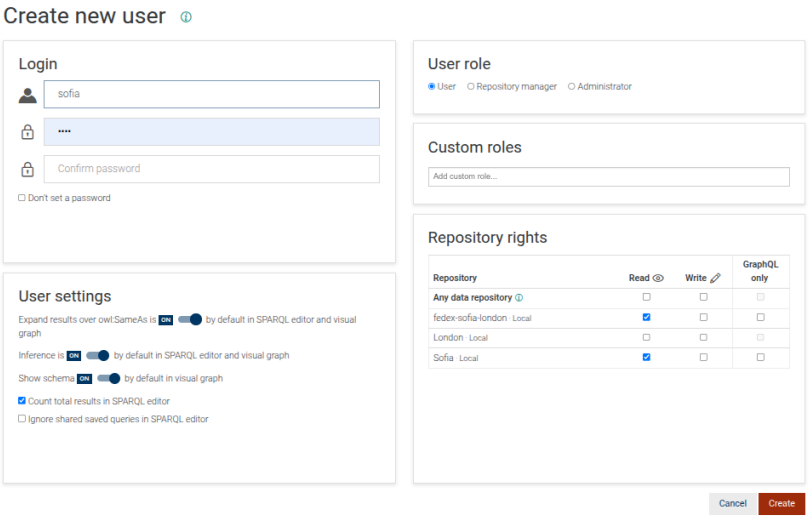
```
SELECT * WHERE {
    ?s ?p ?o .
}
```

4. We can observe that all statements for both Sofia and London are returned as results (here ordered by subject in alphabetical order so as to show results for both):

	s	p	o
1	http://dbpedia.org/resource/Battersea	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
2	http://dbpedia.org/resource/Black_and_White_(EP)	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
3	http://dbpedia.org/resource/Boryana_Rossa	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
4	http://dbpedia.org/resource/City_of_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
5	http://dbpedia.org/resource/FC_Conegliano_German	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
6	http://dbpedia.org/resource/FC_Lyulin	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
7	http://dbpedia.org/resource/FC_Tsarsko_Selo_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
8	http://dbpedia.org/resource/Festa_Holding	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
9	http://dbpedia.org/resource/Greater_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
10	http://dbpedia.org/resource/Helios_Airways	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticipa	http://dbpedia.org/resource/Sofia
11	http://dbpedia.org/resource/Hyde_Park,_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
12	http://dbpedia.org/resource/London	http://www.w3.org/2002/07/owl#sameAs	http://dbpedia.org/resource/London
13	http://dbpedia.org/resource/London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
14	http://dbpedia.org/resource/London,_Ontario	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
15	http://dbpedia.org/resource/Monument_to_1300_Years_of_Bulgaria,	http://www.w3.org/2002/07/owl#differentFrom	http://dbpedia.org/resource/Sofia

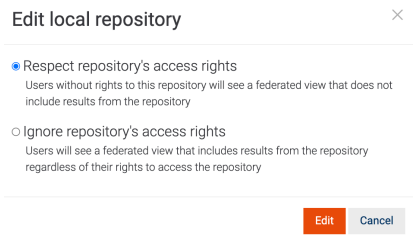
5. Now, to see how this works with GraphDB security enabled, go to *Setup* ▸ *Users and Access*, and set Security to ON.

6. From the same page, create a new user called "sofia" with read rights for the "Sofia" and the FedX repositories:



7. From *Setup* ▸ *Repositories*, click the edit icon of the FedX repository to display its configuration.

8. Click the edit icon of either of the "Sofia" or "London" member repositories. This will open a security setting dialog where you can see that the default setting of each member is to respect the repository's access rights, meaning that if a user has no rights to this repository, they will see a federated view that does not include results from it.



9. Log out as admin and log in as user "sofia".
10. In the SPARQL editor, execute:

```
SELECT * WHERE {
    ?s ?p ?o .
}
```

We can see that only results for the Sofia repository are shown, because the current user has no access to the London repository and the FedX repository is instructed to respect the rights for it.

	s	p	o
1	http://dbpedia.org/resource/Black_and_White_(EP)	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
2	http://dbpedia.org/resource/Boryana_Rossa	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
3	http://dbpedia.org/resource/FC_Conegliano_German	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
4	http://dbpedia.org/resource/FC_Lyulin	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
5	http://dbpedia.org/resource/FC_Tsarsko_Selo_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
6	http://dbpedia.org/resource/Festa_Holding	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
7	http://dbpedia.org/resource/Helios_Airways	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
8	http://dbpedia.org/resource/Monument_to_1300_Years_of_Bulgaria,	http://www.w3.org/2002/07/owl#differentFrom	http://dbpedia.org/resource/Sofia
9	http://dbpedia.org/resource/PFC_Akademik_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
10	http://dbpedia.org/resource/PFC_CSKA_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
11	http://dbpedia.org/resource/PFC_Levski_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
12	http://dbpedia.org/resource/PFC_Lokomotiv_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
13	http://dbpedia.org/resource/PFC_Septemvri_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
14	http://dbpedia.org/resource/PFC_Slavia_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia
15	http://dbpedia.org/resource/Powernet	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#coparticip	http://dbpedia.org/resource/Sofia

11. Log out from the "sofia" user and log back in as admin.
12. Open the edit screen of the FedX repository and set the security of both of its members to ignore the repository's access rights. This means that in the federation, users will see results from the respective repository regardless of their access rights for it.
13. After editing the Sofia and London repositories this way, *Save* the changes in the FedX repository.
14. Log out as admin and log in as user "sofia".
15. In the SPARQL editor, execute:

```
SELECT * WHERE {
    ?s ?p ?o .
}
```

16. This will show that the returned results include statements from both the "Sofia" and the "London" members of the federated repository.

Configuring security for remote endpoints

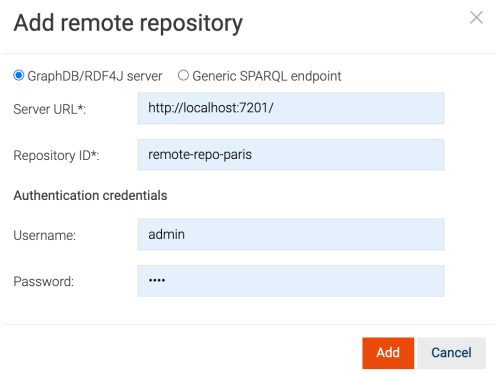
Basic authentication for remote members

GraphDB supports configuration of basic authentication when attaching a remote endpoint. Let's see how this works with the following example:

1. Run a second GraphDB instance on `localhost:7201`. The easiest way to do this is to:
   - Make a copy of your GraphDB distribution.
   - Run it with `graphdb -Dgraphdb.connector.port=7201`.
2. In it, create a repository called "remote-repo-paris" with enabled security and a default user with a username of "admin" and a password of "root".
3. Go to the FactForge SPARQL editor and execute:

```
CONSTRUCT WHERE {
    ?s ?p <http://dbpedia.org/resource/Paris> .
} LIMIT 20
```

4. Download the results as a Turtle file and import them into "remote-repo-paris".
5. Go to the first GraphDB instance on port 7200 and open the "fedx-sofia-london" repository that we created earlier. It already has two members — "Sofia" and "London".
6. In it, include as a member the "remote-repo-paris" we just created:
   a. Select the *GraphDB/RDF4J server* option.
   b. As *Server URL*, enter the URL of the remote repository — `http://localhost:7201/`.
   c. *Repository ID* is the name of the remote repository — `remote-repo-paris`.
   d. *Authentication credentials* are the user and password for the remote repository.
   e. Click the *Add* button.



7. Restart the repository.
8. In the SPARQL editor, execute:

```
SELECT * WHERE {
    ?s ?p ?o .
}
```

We see that all the Paris data from the remote endpoint are available in our FedX repository.

Filter query results

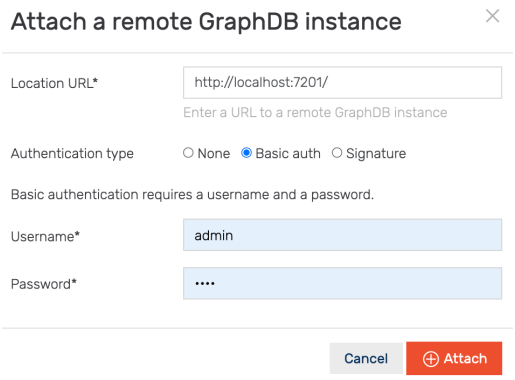
Showing results from 1 to 293 of 293. Query took 16s, today at 15:46.

	s	p	o
1	http://dbpedia.org/resource/American_Cathedral_in_Paris	http://www.w3.org/2002/07/owl#differentFrom	http://dbpedia.org/resource/Paris
2	http://dbpedia.org/resource/Aubervilliers	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/Paris
3	http://dbpedia.org/resource/Battersea	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
4	http://dbpedia.org/resource/Black_and_White_(EP)	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
5	http://dbpedia.org/resource/Boryana_Rossa	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
6	http://dbpedia.org/resource/City_of_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
7	http://dbpedia.org/resource/Culture_of_Paris	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/Paris
8	http://dbpedia.org/resource/FC_Conegliano_German	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
9	http://dbpedia.org/resource/FC_Lyulin	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
10	http://dbpedia.org/resource/FC_Tsarsko_Selo_Sofia	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
11	http://dbpedia.org/resource/Festa_Holding	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
12	http://dbpedia.org/resource/Greater_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
13	http://dbpedia.org/resource/Helios_Airways	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#copartic	http://dbpedia.org/resource/Sofia
14	http://dbpedia.org/resource/Hyde_Park,_London	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/London
15	http://dbpedia.org/resource/List_of_stations_of_the_Paris_Métr	http://www.w3.org/2000/01/rdf-schema#seeAlso	http://dbpedia.org/resource/Paris

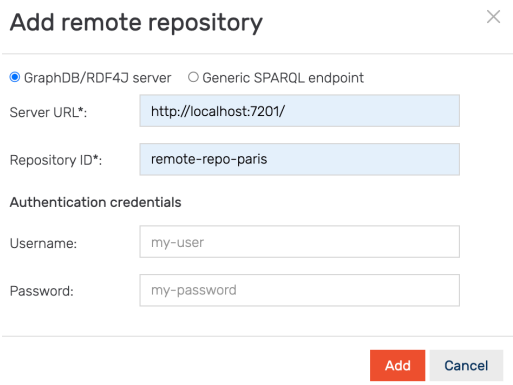
Security of a remote repository from a known location

The context is the same as the previous scenario — two running GDB instances with the second one being secured. The difference is that when the remote repository is a known location, we can configure its security credentials when adding it as a location instead of when adding it as a remote FedX member. Let's see how to do it.

1. Start the same way as in the example above:
   - Run a second GraphDB instance on `localhost:7201`.
   - In it, create a repository called "remote-repo-paris" with enabled security and a default user with a username of "admin" and a password of "root".
   - Import the Paris data in it.
2. In the first GraphDB instance on port 7200, attach "remote-repo-paris" as a *remote location*. For *Authentication type*, select *Basic auth*, and input the user credentials.



3. Again in the 7200 GraphDB instance, open the edit view of the "fedx-sofia-london" repository.
4. In it, include as a member the "remote-repo-paris" from the 7201 port. Note that this time, we are not inputting the security credentials.



5. Restart the FedX repository.
6. In the SPARQL editor, execute:

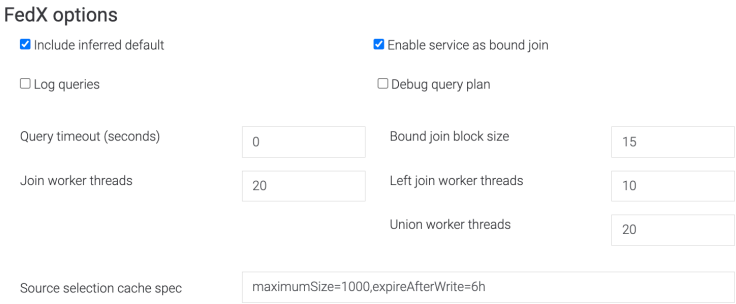
```
SELECT * WHERE {
    ?s ?p ?o .
}
```

Again, we see that all the Paris data from the remote location are available in the FedX repository.

Tip: You can configure *signature authentication* for remote endpoints in the same way.

2.10.4 Configuration parameters

When configuring a FedX repository, several configuration options can be set:



- *Include inferred default*: Whether to include inferred statements. Default is `true`.
- *Enable service as bound join*: Determines whether vectored evaluation using the VALUES clause should be applied for SERVICE expressions. Default is `true`.
- *Log queries*: Enables or disables query logging. Prints the query in the logs. Default is `false`.
- *Log query plan*: Enables or disables query plan logging. Default is `false`.
- *Debug query plan*: The debug mode for the query execution plan. If enabled, the plan is printed in the logs. Default is `false`.
- *Query timeout (seconds)*: Sets the maximum query time in seconds used for query evaluation. Can be set to `0` or less in order to disable query timeouts. If the limit is exceeded, an exception "Query evaluation error: Source selection has run into a timeout" is thrown. Default is `0`.
- *Bound join block size*: The block size for a bound join — that is, the number of bindings that are integrated in a single subquery. Default is `15`.
- *Join worker threads*: The (maximum) number of join worker threads used in the `ControlledWorkerScheduler` for join operations. Default is `20`.

- *Left join worker threads*: The (maximum) number of left join worker threads used in the `ControlledWorkerScheduler` for join operations. Sets the number of threads that can work in parallel evaluating a query with OPTIONAL. Default is `10`.
- *Union worker threads*: The (maximum) number of union worker threads used in the `ControlledWorkerScheduler` for join operations. Sets the number of threads that can work in parallel evaluating a query with UNION. Default is `20`.
- *Source selection cache spec*: Parameters should be passed as `key1=value1,key2=value2,...` in order to be parsed correctly.

  Parameters that can be passed:

  - `recordStats` (boolean)
  - `initialCapacity` (int)
  - `maximumSize` (long)
  - `maximumWeight` (long)
  - `concurrencyLevel` (int)
  - `recordStats` (boolean)
  - `refreshDuration` (long)
  - `expireAfterWrite` (TimeUnit/long)
  - `expireAfterAccess` (TimeUnit/long)
  - `refreshAfterWrite` (TimeUnit/long)

2.10.5 Limitations

Some limitations of GraphDB's current FedX federation implementation are:

- DESCRIBE queries are not supported.
- FedX is not stable with queries of the type `{?s ?p ?o} UNION {?s ?p1 ?o} FILTER (xxx)`.
- The federation currently only works with remote repositories. Everything goes through HTTP, which is slower compared to direct access to local repositories.
- Queries with a Cartesian product or cyclic connections are not stable due to connections that are still open and to blocked threads.
- There is a small possibility of threads being blocked on complex queries due to implementation flows in parallelization.

2.11 Graph replacement optimization

GraphDB offers an optimized graph replacement algorithm which makes graph updates faster in situations where the new graph partially overlaps with data in the old one.

Clearing an old graph and then importing the new information can be inefficient. If the two operations are handled separately, it is impossible to determine whether a statement will also be present in the new graph and thus preserve it instead of deleting it. The same applies for preserving connectors or inferring statements.

As long as the replacement is done in a single transaction, the graph replacement optimization will be applied to avoid such inefficiencies. The property `graphdb.replace.graph.batch.size` controls the batch size of the buffer, and has a default value of `10000` statements.

The graph replacement optimization algorithm implements the following logic:

1. Check transaction contents. If the transaction includes a graph replacement, proceed.

2. Check if any of the graphs to be replaced are valid and if any of them have data. If so, store their identifiers in a list.
3. While processing transaction statements for insertion, if their context (graph) matches an identifier from the list, store them inside a tracker.
4. While clearing the graph to be replaced, if it is not mentioned in the tracker, directly delete all its contents.
5. If a graph is mentioned in the tracker, iterate over its triples.
6. Triples in the replacement graph that are also in the tracker are preserved. Otherwise, they are deleted.

Deletions may trigger re-inference and are a more costly process than the check described in the algorithm. Therefore, in some test cases due to the optimization users can observe a speedup of up to 200%.

Here is an example of an update that will use the replacement optimization algorithm:

```
curl -X PUT -H "Content-Type: application/x-trig" --data-binary '@test_modified.trig'\
    'http://localhost:7200/repositories/test/rdf-graphs/service?graph=http://example.org/optimizations/
↪replacement'
```

By contrast, the following approach will **not** use the optimization since it performs the replacement in two separate steps:

```
curl -X POST -H 'Content-Type: application/sparql-update'\
    --data-binary 'CLEAR GRAPH <http://example.org/optimizations/replacement>'\
    'http://localhost:7200/repositories/test/statements'
```

```
curl -X POST -H "Content-Type: application/x-trig" --data-binary '@test_modified.trig'\
    'http://localhost:7200/repositories/test/statements'
```

Note: The replacement optimization described here applies to all forms of transactions. It will be triggered by standard PUT requests, such as the ones in the example, but also by SPARQL INSERT queries containing the `http://www.ontotext.com/replaceGraph` predicate, such as `<http://any/subject> <http://wwww.ontotext.com/replaceGraph> <http://example.org/graph>`

2.12 Managing repositories and locations via the GraphDB REST API

GraphDB's REST API lets you perform basic location and repository management. This includes connecting to remote GraphDB instances (locations), activating a location, and managing local and remote repositories.

This page shows how to execute these requests with the `curl` command line utility. Most modern programming languages can execute HTTP requests like this either natively or with an available library. This lets you integrate these capabilities into your custom applications.

Note: The GraphDB command line utility *storage-tool* gives you several ways to work with GraphDB repositories from your operating system's command line interface.

2.12.1 Managing repositories

Most repository management queries can either take the following set of attributes as an argument or return them as a response:

- `externalUrl` (string): The URL where the repository can be accessed by an external service
- `id` (string): The repository ID
- `local` (boolean): `True` if the repository is local (on the same machine as Workbench)
- `location` (string): If remote, the repository's location
- `title` (string): The repository title
- `type` (string): Repository type — `graphdb`, `ontop`, or `fedx`
- `unsupported` (boolean): `True` if the repository is unsupported
- `writable` (boolean): `True` if the repository is writable
- `readable` (boolean): `True` if the repository is readable
- `uri` (string): The GraphDB location URL

2.12.1.1 Create a repository

GraphDB lets you create a repository with an API call that sends RDF data of the configuration information. (If using GraphDB Free, your instance can only have five repositories; see *Creating a repository* for more on this.) You can try this yourself with the following example.

1. Download the sample repository config file `repo-config.ttl`.
2. Send the file with a `POST` request using the following `curl` command:

```
curl -X POST\
    http://localhost:7200/rest/repositories\
    -H 'Content-Type: multipart/form-data'\
    -F "config=@repo-config.ttl"
```

2.12.1.2 List repositories

The following `curl` command lists all repositories by sending a `GET` request to the API:

```
curl http://localhost:7200/rest/repositories\
    -H 'Accept: application/json'
```

The output shows one repository named `repo1`.

```
[
   {
      "id":"repo1",
      "title":"my repository number one",
      "uri":"http://localhost:7200/repositories/repo1",
      "type":"free",
      "sesameType":"graphdb:SailRepository",
      "location":"",
      "readable":true,
      "writable":true,
      "local":true
   }
]
```

You can retrieve information about all the repositories at a remote location by including that location as a parameter:

```
curl http://localhost:7200/rest/repositories?location=http://192.168.0.79:7200
```

2.12.1.3 Retrieve information about one repository

The command to list repositories shown in the previous section will also list some information about all the repositories stored on the GraphDB instance at that URL. To retrieve more detailed information about a single repository, add the repository ID to the end of the URL that is used to list repositories:

```
curl http://localhost:7200/rest/repositories/repo1
```

As when *listing repositories*, adding a `?location` parameter to the URL lets you retrieve information about one repository at the specified location.

2.12.1.4 Count the triples in a repository

Adding `/size` to the URL that request information about a repository asks for its size, measured in triples:

```
curl http://localhost:7200/rest/repositories/repo1/size
```

As when *listing repositories*, adding a `?location` parameter to the URL lets you count the triples in a repository at the specified location.

2.12.1.5 Restart a repository

An HTTP `POST` request that names a specific repository and adds `/restart` at the end of that URL instructs the server to restart that repository:

```
curl -X POST http://localhost:7200/rest/repositories/repo1/restart
```

As when *listing repositories*, adding a `?location` parameter to the URL will restart the named repository at that location.

2.12.1.6 Delete a repository

An HTTP `DELETE` request that names a specific repository instructs the server to delete that repository:

```
curl -X DELETE http://localhost:7200/rest/repositories/repo1
```

As when *listing repositories*, adding a `?location` parameter to the URL will delete the named repository at that location, assuming that the appropriate write permissions are in place.

2.12.1.7 Edit a repository's configuration

To edit a repository's configuration, do an HTTP `PUT` of the new configuration stored as JSON, as shown in the following model.

Note: When editing the configuration of a local repository, omit the "location" parameter.

```
curl -X PUT <base_url>/rest/repositories/<repo_id> -H 'Accept: application/json' -H 'Content-Type:
↪application/json' -d '
    {
        "id": "<repo_id>",
```

(continues on next page)

(continued from previous page)

```
        "location": "<location_uri>",
        "title": "<repo_title>",
        "type": "graphdb",
        "sesameType":"graphdb:SailRepository",
        "params":{
            "queryTimeout":{
                "name":"queryTimeout",
                "label":"Query timeout (seconds)",
                "value":"0"
            },
            "cacheSelectNodes":{
                "name":"cacheSelectNodes",
                "label":"Cache select nodes",
                "value":"true"
            },
            "rdfsSubClassReasoning":{
                "name":"rdfsSubClassReasoning",
                "label":"RDFS subClass reasoning",
                "value":"true"
            },
            "validationEnabled":{
                "name":"validationEnabled",
                "label":"Enable the SHACL validation",
                "value":"true"
            },
            "ftsStringLiteralsIndex":{
                "name":"ftsStringLiteralsIndex",
                "label":"FTS index for xsd:string literals",
                "value":"default"
            },
            "shapesGraph":{
                "name":"shapesGraph",
                "label":"Named graphs for SHACL shapes",
                "value":"http://rdf4j.org/schema/rdf4j#SHACLShapeGraph"
            },
            "parallelValidation":{
                "name":"parallelValidation",
                "label":"Run parallel validation",
                "value":"true"
            },
            "checkForInconsistencies":{
                "name":"checkForInconsistencies",
                "label":"Enable consistency checks",
                "value":"false"
            },
            "performanceLogging":{
                "name":"performanceLogging",
                "label":"Log the execution time per shape",
                "value":"false"
            },
            "disableSameAs":{
                "name":"disableSameAs",
                "label":"Disable owl:sameAs",
                "value":"true"
            },
            "ftsIrisIndex":{
                "name":"ftsIrisIndex",
                "label":"FTS index for full-text indexing of IRIs",
                "value":"en"
            },
            "entityIndexSize":{
```

(continues on next page)

(continued from previous page)

```
                "name":"entityIndexSize",
                "label":"Entity index size",
                "value":"10000000"
            },
            "dashDataShapes":{
                "name":"dashDataShapes",
                "label":"DASH data shapes extensions",
                "value":"true"
            },
            "queryLimitResults":{
                "name":"queryLimitResults",
                "label":"Limit query results",
                "value":"0"
            },
            "throwQueryEvaluationExceptionOnTimeout":{
                "name":"throwQueryEvaluationExceptionOnTimeout",
                "label":"Throw exception on query timeout",
                "value":"false"
            },
            "member":{
                "name":"member",
                "label":"FedX repo members",
                "value":[]
            },
            "storageFolder":{
                "name":"storageFolder",
                "label":"Storage folder",
                "value":"storage"
            },
            "validationResultsLimitPerConstraint":{
                "name":"validationResultsLimitPerConstraint",
                "label":"Validation results limit per constraint",
                "value":"1000"
            },
            "enablePredicateList":{
                "name":"enablePredicateList",
                "label":"Enable predicate list index",
                "value":"true"
            },
            "transactionalValidationLimit":{
                "name":"transactionalValidationLimit",
                "label":"Transactional validation limit",
                "value":"500000"
            },
            "ftsIndexes":{
                "name":"ftsIndexes",
                "label":"FTS indexes to build (comma delimited)",
                "value":"default, iri, en"
            },
            "logValidationPlans":{
                "name":"logValidationPlans",
                "label":"Log the executed validation plans",
                "value":"false"
            },
            "imports":{
                "name":"imports",
                "label":"Imported RDF files('\'';'\'' delimited)",
                "value":""
            },
            "isShacl":{
                "name":"isShacl",
```

(continues on next page)

(continued from previous page)

```
            "label":"Enable SHACL validation",
            "value":"false"
        },
        "inMemoryLiteralProperties":{
            "name":"inMemoryLiteralProperties",
            "label":"Cache literal language tags",
            "value":"true"
        },
        "ruleset":{
            "name":"ruleset",
            "label":"Ruleset",
            "value":"rdfsplus-optimized"
        },
        "readOnly":{
            "name":"readOnly",
            "label":"Read-only",
            "value":"false"
        },
        "enableLiteralIndex":{
            "name":"enableLiteralIndex",
            "label":"Enable literal index",
            "value":"true"
        },
        "enableFtsIndex":{
            "name":"enableFtsIndex",
            "label":"Enable full-text search (FTS) index",
            "value":"false"
        },
        "defaultNS":{
            "name":"defaultNS",
            "label":"Default namespaces for imports('\'';'\'' delimited)",
            "value":""
        },
        "enableContextIndex":{
            "name":"enableContextIndex",
            "label":"Enable context index",
            "value":"false"
        },
        "baseURL":{
            "name":"baseURL",
            "label":"Base URL",
            "value":"http://example.org/owlim#"
        },
        "logValidationViolations":{
            "name":"logValidationViolations",
            "label":"Log validation violations",
            "value":"false"
        },
        "globalLogValidationExecution":{
            "name":"globalLogValidationExecution",
            "label":"Log every execution step of the SHACL validation",
            "value":"false"
        },
        "entityIdSize":{
            "name":"entityIdSize",
            "label":"Entity ID size",
            "value":"32"
        },
        "repositoryType":{
            "name":"repositoryType",
            "label":"Repository type",
```

(continues on next page)

(continued from previous page)

```
                "value":"file-repository"
            },
            "eclipseRdf4jShaclExtensions":{
                "name":"eclipseRdf4jShaclExtensions",
                "label":"RDF4J SHACL extensions",
                "value":"true"
            },
            "validationResultsLimitTotal":{
                "name":"validationResultsLimitTotal",
                "label":"Validation results limit total",
                "value":"1000000"}
            }
        }
    }'
```

Tip: Adjust parameters with new values except for `<repo_id>` in order to edit the current repository configuration.

2.12.2 Managing locations

Tip: Throughout this document, the two example instances will be referred to with the following URLs:

- `http://192.0.2.1:7200/` for the first instance
- `http://192.0.2.2:7200/` for the second instance

Adjust the URLs according to the IP or hostname values of your own machines.

2.12.2.1 Attach a location

This section describes how you can attach your locally running instance of GraphDB to another one on the same network.

The following `curl` command attaches the remote `http://192.0.2.2:7200` GraphDB instance to the local `http://192.0.2.1:7200` one by sending a `PUT` request to the API running on the local instance:

```
curl -X PUT http://192.0.2.1:7200/rest/locations\
    -H 'Content-Type:application/json'\
    -d '{
        "uri": "http://192.0.2.2:7200/",
        "username": "admin",
        "password": "root"
    }'
```

Note: The `username` and `password` may not be necessary, depending on the settings of the remote location.

2.12.2.2 List locations

Use the following `curl` command to list all attached locations by sending a `GET` request to the API:

```
curl http://192.0.2.1:7200/rest/locations\
    -H 'Accept: application/json'
```

The output shows one local and one remote location:

```
[
   {
      "system" : true,
      "errorMsg" : null,
      "active" : false,
      "defaultRepository" : null,
      "local" : true,
      "username" : null,
      "uri" : "",
      "password" : null,
      "label" : "Local"
   },
   {
      "system" : false,
      "errorMsg" : null,
      "active" : true,
      "defaultRepository" : null,
      "local" : false,
      "username" : "admin",
      "uri" : "http://192.0.2.1:7200/",
      "password" : "root",
      "label" : "Remote (http://192.0.2.1:7200/)"
   }
]
```

2.12.2.3 Detach a location

Use the following `curl` command to detach a remote location by sending a `DELETE` request to the API:

- To detach the remote location `http://192.0.2.1:7200/`:

  ```
  curl -X DELETE http://192.0.2.1:7200/rest/locations\
      -H 'Content-Type:application/json'\
      -d uri=http://192.0.2.2:7200/
  ```

Important: Detaching a location simply removes it from Workbench and will **not** delete any data. A detached location can be re-attached at any point.

2.12.3 Further reading

For a full list of request parameters and more information about sending requests, check the REST API documentation within the GraphDB Workbench accessible from the *Help* ▸ *REST API* menu.

CHAPTER

THREE

LOADING AND EXPLORING DATA

3.1 Loading data

GraphDB offers multiple interfaces for loading RDF data in your repository. Each interface supports the loading of *any RDF format*, but the different interfaces have different limitations and advantages.

You can access these interfaces through Workbench or through the Import RDF tool. You can also load RDF data via HTTP with `curl`.

Table 1: GraphDB's data loading interfaces

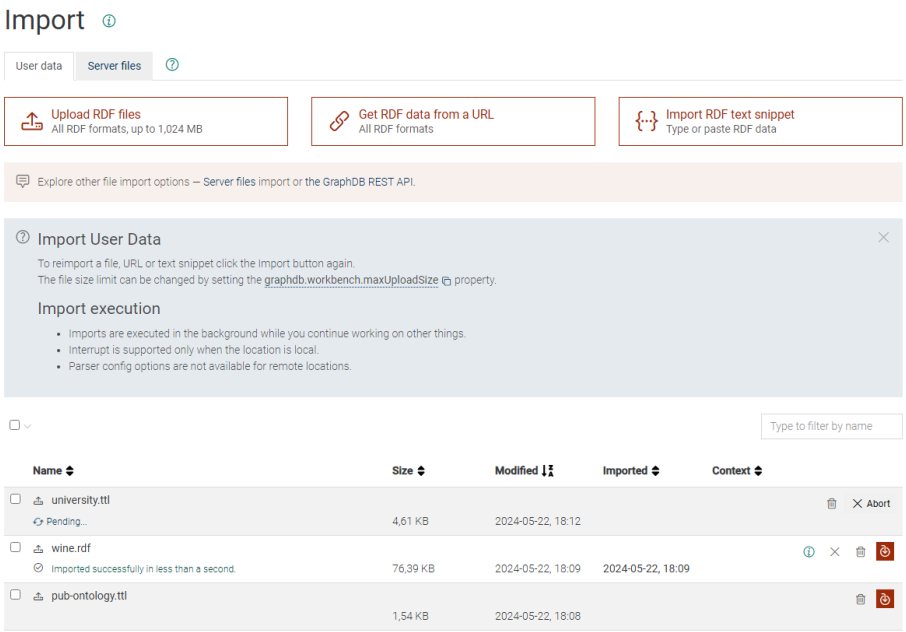
Interface	Use cases	Mode	Speed
Workbench import of a local RDF file	Small files, limited to 1GB by default	Online parallel	Moderate speed
Workbench import of a remote RDF file	Small files, limited to 1GB by default	Online parallel	Moderate speed
Workbench import of a text snippet	Small text snippets	Online parallel	Moderate speed
Workbench import of a server file	No limits on the file size	Online parallel	Fast, ignoring all HTTP protocol overheads
SPARQL endpoint	No limits on the file size	Online parallel	Moderate speed
ImportRDF Load	Batch import of very big files	Initial offline import with no plugins	Fast, with small speed degradation
ImportRDF Preload	Import huge datasets with no inference	Initial offline import with no inference and plugins	Ultra-fast without speed degradation

Tip: It's often useful to use GraphDB's *rdfvalidator* command line utility to check that an RDF file parses properly before attempting to load it.

3.1.1 Loading data using Workbench

The Workbench *Import* view hosts many of the interfaces used for loading RDF data into your repository, such as loading data:

- from local files
- from a remote URL (with a format extension or by specifying the data format)
- by pasting the RDF data in the *Text area* tab
- from files on the server where Workbench is located



You can also load RDF data with an `INSERT` update request in the *SPARQL editor*.

All import methods support asynchronous running of the import tasks, with the exception of the text area import, which is intended for very fast and simple imports. Currently, only one import task of a type is executed at a time, while the others wait in the queue as pending.

Note: Since Workbench performs the parsing when importing data into a local repository, GraphDB supports interruption and additional settings. When importing data into a remote location, the data is sent to the remote endpoint, and the parsing and loading are performed there.

There are two tabs inside the view — *User data*, for RDF data provided by the user, and *Server files*, for files that are already hosted on the server. Both tabs provide a file name filter to narrow down the list of available files.

Note: When importing data from local files, a remote URL, or by pasting a text snippet, the files are first uploaded to the server. Once uploaded to the server, they can be imported into the repository.

3.1.1.1 Importing local RDF files

Go to *Import* ‣ *User data* ‣ *Upload RDF files*.

This lets you select, configure, and import data from *various RDF formats*.

Note: This method supports files of a limited size. The default is 200 megabytes, and is controlled by the `graphdb.workbench.maxUploadSize` property where it is set in bytes (`-Dgraphdb.workbench.maxUploadSize=1073741824`).

Loading data from your local machine directly streams the file to the RDF4J *statements* endpoint:

1. Click the button to browse files for uploading.
2. When the files appear in the table, either import a file by clicking *Import* on its line or select multiple files and click *Import* in the header.
3. The *Import settings* dialog box appears in case you want to reset any other parameters.

3.1.1.2 Importing remote RDF files

Go to *Import* ‣ *User data* ‣ *Get RDF data from a URL*.

You can import from a URL with RDF data. Each endpoint that returns RDF data can be used.

If the URL has an extension, GraphDB uses that to detect the correct data type (for example, `.ttl` for a Turtle file and `.rdf` for an RDF/XML file; see *RDF formats* for a complete list). Otherwise, you have to provide the Data Format parameter, which is sent as Accept header to the endpoint and then to the import loader.

3.1.1.3 Importing RDF data from a text snippet

Go to *Import* ‣ *User data* ‣ *Import RDF text snippet*.

You can import data by typing or pasting it directly in the text area. This has the same effect as uploading a small RDF file and importing it.

3.1.1.4 Importing server files

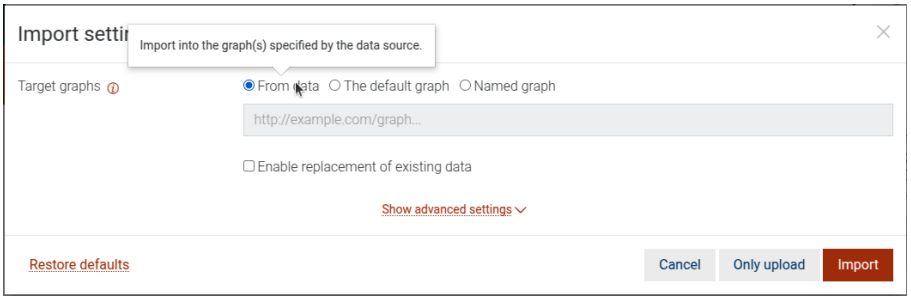
Go to *Import* ‣ *Server files*.

The server files import allows you to load files of arbitrary sizes. Its limitation is that the files must be put in a specific directory. (Symbolic links are supported.) By default, it is `$user.home/graphdb-import/` that you need to create beforehand.

If you'd rather use a different directory location, see the `graphdb.workbench.importDirectory` *system property*. GraphDB scans the directory recursively and all files with a semantic MIME type are visible in the *Server files* tab.

3.1.1.5 Import settings

The settings for each import are saved in case you want to re-import a file. You can see them in the dialog box that opens after you have uploaded a document and clicked *Import*:



- *Target graphs* — when specified, the graphs where the imported data should be stored. *RDF formats* such as TriG and N-Quads may list specific named graphs where some or all of the triples should be stored, but formats such as Turtle and RDF/XML do not support this. Formats in the latter category are treated as if they specify the default graph as the target.
  - From data — Imports data into the graph (or graphs, for formats that can specify named graphs) specified by the data source.
  - The default graph — Imports all data into the default graph.
  - Named graph — Imports everything into a named graph that you specify on this dialog box.
- *Enable replacement of existing data* — Let the imported data replace existing data in the repository.
  - *Replaced graph(s)* — All specified graphs will be cleared before the import is run. If a graph name ends in `*`, it will be treated as a prefix matching all named graphs starting with that prefix excluding the `*`. This option provides the most flexibility when the target graphs are determined from data.

  - *I understand that data in the replaced graphs will be cleared before importing new data* — this option must be checked when the data replacement is enabled.

- *JSON-LD context* — Only visible when importing JSON-LD or NDJSON-LD files. Specifies external context as an URL. Only whitelisted URLs can be used.

> **Tip:** The whitelist is a comma-separated list of URLs used to validate any context URLs in imported JSON-LD or specified by the user for export. The wildcard * allows for fine-grained control, enabling administrators to specify a set of URLs, including entire directories. Each entry in the list represents a source that is considered safe for JSON-LD and NDJSON-LD operations.
>
> ```
> # Sets whitelist for JSON-LD resources
> graphdb.jsonld.whitelist = https://example.com/my/jsonld/*, file:///usr/local/my/jsonld/
> ↪*
> ```

Advanced settings:

- *Base IRI* — the base IRI against which to resolve any relative IRIs found in the uploaded data. When data does not contain relative IRIs, this field may be left empty.
- *Preserve BNode IDs*: whether to use the blank node IDs found in the imported file or to assign their own internal blank node identifiers.
- *Fail parsing if datatypes are not recognized*: whether to fail parsing if datatypes are unknown.
- *Verify recognized datatypes*: verify that the values of the datatype properties in the file are valid.
- *Normalize recognized datatypes values*: whether recognized datatypes need to have their values be normalized.
- *Fail parsing if languages are not recognized*: whether to fail parsing if languages are unknown.
- *Verify language based on a given set of definitions for valid languages*: whether languages tags are to be verified.
- *Normalize recognized language tags*: whether languages need to be normalized, and to which format they should be normalized.
- *Should stop on error*: whether to ignore non-fatal errors.
- *Force serial pipeline*: use the serial pipeline when importing data.

The Workbench *Import* view lists a table of previously imported files with the sortable columns *Name*, *Size*, *Modified*, *Imported*, and *Context*. Clicking the checkbox to the left of a table displays an *Import* button where you can re-import any checked files and a *Remove* button that lets you remove any checked files from the table. To the right of the *Import* button, a dropdown menu ⌄ lets you re-import any selected items without changing their import settings.

3.1.1.6 Import data with an INSERT query using the SPARQL endpoint

You can also insert triples into a graph with an `INSERT` update request in the *SPARQL editor*.

3.1.2 Loading data using the ImportRDF tool

The ImportRDF tool is designed for offline loading of datasets. It cannot load data into a running server. An offline tool provides optimal performance for loading large amounts of RDF data by directly serializing them into GraphDB's internal indexes and producing a ready-to-use repository.

The ImportRDF tool resides in the `bin` folder of the GraphDB *distribution directory*. It loads data in a new repository created from Workbench or from the standard configuration Turtle file found in `configs/templates`, or it uses an existing repository. In the latter case, the repository data is automatically overwritten.

Important: The ImportRDF tool cannot be used in a *cluster setup* as it would break the cluster consistency.

3.1.2.1 Load vs Preload

The ImportRDF tool supports two sub-commands: Load and Preload. (These are supported as separate commands in GraphDB versions 9.x and older.)

Despite the many similarities between Load and Preload (such as how both do parallel offline transformation of RDF files into a GraphDB image) there are substantial differences in their implementation. Load uses an algorithm very similar to online data loading. As the data variety grows, the loading speed drops because the page splits and the tree is rebalancing. After a continuous data load, the disk image becomes fragmented in the same way as it would happen if the RDF files were imported into the engine.

The use of Preload eliminates the performance drop by implementing a two-phase load. In the first phase, all RDF statements are processed in-memory in chunks, which are later flushed on the disk as many GraphDB images. Then, all sorted chunks are merged into a single non-fragmented repository image with a merge join algorithm. Because of this, the Preload sub-command requires almost twice as much disk space to complete the import.

Preload does not perform inference on the data.

Warning: During the bulk load, the GraphDB plugins are ignored in order to speed up the process. Afterwards, when the server is started, the plugin data can be rebuilt.

Note: The ImportRDF tool supports *various RDF formats*, `.zip` and `.gz` files, and directories.

3.1.2.2 Command line options

These are covered in the following sections of the *GraphDB command line tools* documentation page:

- *Load command line options*
- *Preload command line options*

3.1.2.3 Loading data

There are two ways to load data with the ImportRDF tool.

Into a repository created from Workbench

1. Configure the ImportRDF repository location by setting the property `graphdb.home.data` in `conf/graphdb.properties`. If no property is set, the default repository location will be the `data` subdirectory of the GraphDB *distribution directory*.
2. Start GraphDB.
3. In a browser, open the Workbench web application at http://localhost:7200. If necessary, replace `localhost` and the `7200` port number as appropriate.
4. Go to *Setup* ▸ *Repositories*.
5. *Create* and *configure* a repository.
6. Stop GraphDB.
7. Start the bulk load with the following command:

   ```
   $ <graphdb-dist>/bin/importrdf load -f -i <repo-id> -m parallel <RDF data file(s)>
   ```

   or if using the `preload` sub-command:

   ```
   $ <graphdb-dist>/bin/importrdf preload -f -i <repo-id> <RDF data file(s)>
   ```

8. Start GraphDB.

Into a repository created from a config file

1. Stop GraphDB.
2. Configure the ImportRDF repository location by setting the property `graphdb.home.data` in `conf/graphdb.properties`. If no property is set, the default repository location will be the `data` subdirectory of the GraphDB *distribution directory*.
3. Create a *configuration file*.
4. Start the bulk load with the following command:

   ```
   $ <graphdb-dist>/bin/importrdf load -c <repo-config.ttl> -m parallel <RDF data file(s)>
   ```

   or if using the `preload` sub-command:

   ```
   $ <graphdb-dist>/bin/importrdf preload -f -c <repo-config.ttl> <RDF data file(s)>
   ```

5. Start GraphDB.

Into a named graph

Loading an N-Quads file with the ImportRDF tool will also load the specified triples to the named graphs listed with them in the file. When loading triples to a named graph, no special parameters or configuration files are necessary.

Start the bulk load with the following command:

```
$ <graphdb-dist>/bin/importrdf load -f -i <repo-id> -m parallel <nquads data file(s)>
```

3.1.2.4 Repository configuration template

This is an example configuration template using a minimal parameters set. You can add more optional parameters from the `configs/templates` example:

Tip: To see what configuration data is stored in a GraphDB repository, go to *Repositories* and use the *Download repository configuration as Turtle* icon to see the configuration of any of the listed repositories in a file named `repositoryname-config.ttl`.

```
# Configuration template for a GraphDB repository

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rep: <http://www.openrdf.org/config/repository#>.
@prefix sr: <http://www.openrdf.org/config/repository/sail#>.
@prefix sail: <http://www.openrdf.org/config/sail#>.
@prefix graphdb: <http://www.ontotext.com/config/graphdb#>.

[] a rep:Repository ;
    rep:repositoryID "repo-test-1" ;
    rdfs:label "My first test repo" ;
    rep:repositoryImpl [
    rep:repositoryType "graphdb:SailRepository" ;
    sr:sailImpl [
        sail:sailType "graphdb:Sail" ;

            # ruleset to use
            graphdb:ruleset "empty" ;

            # disable context index(because my data do not uses contexts)
            graphdb:enable-context-index "false" ;

            # indexes to speed up the read queries
            graphdb:enablePredicateList "true" ;
            graphdb:enable-literal-index "true" ;
            graphdb:in-memory-literal-properties "true"
    ]
].
```

3.1.2.5 Tuning load

The ImportRDF tool accepts Java command line options using `-D`. Supply them before the sub-command as follows:

```
$ <graphdb-dist>/bin/importrdf -Dgraphdb.inference.concurrency=6 load -c <repo-
config.ttl> -m parallel <RDF data file(s)>
```

Use the following options to fine-tune the behavior of the `load` sub-command:

- `-Dgraphdb.inference.buffer`: the *buffer size* (the number of statements) for each stage. Defaults to 200,000 statements. You can use this parameter to tune the memory usage and the overhead of inserting data:
  - less buffer size reduces the memory required
  - bigger buffer size reduces the overhead as the operations performed by threads have a lower probability of waiting for the operations on which they rely, and the CPU is intensively used most of the time
- `-Dgraphdb.inference.concurrency`: the *number of inference threads* in parallel mode. The default value is the number of cores of the machine processor. A bigger pool theoretically means faster load if there are enough unoccupied cores and the inference does not wait for the other load stages to complete.

3.1.2.6 Tuning preload

The preload sub-command accepts the following options to fine-tune its operation:

- `--chunk`: the size of the in-memory buffer to sort RDF statements before flushing it to the disk. A bigger chunk consumes additional RAM and reduces the number of chunks to merge. We recommend the default value of 20 million for datasets of up to 20 billion RDF statements.
- `--iterator-cache`: the number of triples to cache from each chunk during the merge phase. A bigger value is likely to eliminate the I/O wait time at the cost of more RAM. We recommend the default value of 64,000 for datasets of up to 20 billion RDF statements.
- `--parsing-tasks`: the number of parsing tasks, which controls how many parallel threads parse the input files.
- `--queue-folder`: the file system location, where all temporary chunks are stored.

3.1.2.7 Resuming data loading with preload

The loading of a huge dataset is a long batch process, and every run may take many hours. Preload supports resuming of the process if something goes wrong such as running out of disk space or memory and the loading is terminated abnormally. In this case, the data processing will restart from an intermediate restore point instead of from the beginning. The data collected for the restore points is sufficient to initialize all internal components correctly and to continue the load normally at that moment, thus saving time. The following options can be used to configure the resumption of data loading:

- `--interval`: sets the recovery point interval in seconds. The default is 3,600s (60min).
- `--restart`: if set to `true`, the loading will start from the beginning, ignoring an existing recovery point. The default is `false`.

Updating data in GraphDB is done via smart updates using server-side SPARQL templates.

3.1.3 Updating data with SPARQL templates

3.1.3.1 Overview

Updating the content of RDF documents can be tricky because of the lack of a fixed schema or standard notion for management of multi-document graphs. There are two widely employed strategies when it comes to managing RDF documents: either storing each RDF document in a single named graph or storing each RDF document as a collection of triples where multiple RDF documents exist in the same graph.

The single RDF document per named graph is easy to update. You can simply replace the content of the named graph with the updated document, and GraphDB provides an optimization to do that efficiently. However, when there are multiple documents in a graph and a single document needs to be updated, the old content of the document must be removed first. This is typically done using a handcrafted SPARQL update that deletes only the triples that define the document. This update needs to be the same on every client that updates data in order to get consistent behavior across the system.

GraphDB solves this by enabling smart updates using server-side SPARQL templates. Each template corresponds to a single document type and defines the SPARQL update that must be executed in order to remove the previous content of the document.

To initiate a smart update, the user provides the IRI identifying the template (that is, the document type) and the IRI identifying the document. The new content of the document is then simply added to the database in any of the supported ways such as graph, SPARQL `INSERT`, or add statements.

Replace graph

A document (the smallest update unit) is defined as the contents of a named graph. Thus, to perform an update, you need to provide the following information:

- The IRI of the named graph — the document ID
- The new RDF contents of the named graph — the document contents

DELETE/INSERT template

A document is defined as all triples for a given document identifier according to a predefined schema. The schema is described as a SPARQL DELETE/INSERT template that can be filled from the provided data at update time. The following must be present at update time:

- The SPARQL template update (must be predefined, not provided at update time)
  - Can be a `DELETE WHERE` update that only deletes the previous version of the document and the new data is inserted as is.
  - Can be a `DELETE INSERT WHERE` update that deletes the previous version of the document and adds additional triples such as timestamp information.
- The IRI of the updated document
- The new RDF contents of the updated document

3.1.3.2 Transport mechanisms

The transport mechanism defines how users send RDF update data to GraphDB. Two mechanisms are supported — direct access and indirect access via the *Kafka Sink connector*.

Direct access

Direct access is a direct connection to GraphDB using the RDF4J API as well as any GraphDB extensions to that API such as using SPARQL or deleting or adding individual triples.

Replace graph

When a replace graph smart update is sent directly to GraphDB, the user does not need to do anything special — for example, a simple `CLEAR GRAPH` followed by INSERT in the same graph.

DELETE/INSERT template

Unlike replace graph, this update mechanism needs a predefined *SPARQL template* that can be referenced at update time. Once a template has been defined, the user can request its use by inserting a system triple.

Let's see how such a template can be used.

1. Create a repository.
2. In the SPARQL editor, add the following data about two employees in a factory and their salaries:

```
PREFIX factory: <http://factory/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
INSERT DATA {
    factory:1 rdf:type factory:Factory .
```

(continues on next page)

(continued from previous page)

```
    factory:John <http://factory/hasSalary> 10000 ;
                 <http://factory/worksIn> factory:1 .
    factory:Luke <http://factory/hasSalary> 10000 ;
                 <http://factory/worksIn> factory:1 .
}
```

3. If we run a simple `SELECT` query to get all information about John,

```
SELECT * WHERE {
    <http://factory/John> ?p ?o .
}
```

we get the following result:

	p	o
1	http://factory/hasSalary	"10000"^^xsd:integer
2	http://factory/worksIn	http://factory/1

4. Again in the SPARQL editor, create and execute the following template:

```
INSERT DATA {
<http://example.com/my-template> <http://www.ontotext.com/sparql/template> '''
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX factory: <http://factory/>
    DELETE {
        ?worker factory:hasSalary ?oldSalary .
    } INSERT {
        ?id factory:updatedOn ?now
    } WHERE {
        ?id rdf:type factory:Factory .
        ?worker factory:worksIn ?id .
        ?worker factory:hasSalary ?oldSalary .
        BIND(now() as ?now)
    }
    '''
}
```

5. Next, execute a smart update to the RDF data, changing the employees' salaries:

```
PREFIX onto: <http://www.ontotext.com/>
PREFIX factory: <http://factory/>
insert data {
    onto:smart-update onto:sparql-template <http://example.com/my-template> ;
                      onto:template-binding-id factory:1 .
    factory:John factory:hasSalary 20000 .
    factory:Luke factory:hasSalary 20000 .
}
```

6. Now let's see how the data has changed. Run the following query again:

```
SELECT * WHERE {
    <http://factory/John> ?p ?o .
}
```

We can see that John's salary has increased.

	p	o
1	http://factory/hasSalary	"20000"^^xsd:integer
2	http://factory/worksIn	http://factory/1

Indirect access via Kafka Sink connector

In this mode, the user pushes update messages to Kafka and the Kafka Sink Connector performs the updates. Users and consumers must agree on the following:

- A given Kafka topic is configured to accept RDF updates in a predefined update type and format.
- The types of updates that can be performed are: replace graph, DELETE/INSERT template, or simple add.
- The format of the data must be one of *the supported RDF formats*.

For more details, see *Kafka Sink connector*.

The following describes how to perform the various classes of updates.

Replace graph

- The Kafka topic is configured for replace graph.
- The Kafka key defines the named graph to update.
- The Kafka value defines the contents of the named graph.

DELETE/INSERT template

- The Kafka topic is configured for a specific template.
- The Kafka key defines the document IRI.
- The Kafka value defines the new contents of the document.

Simple add

- The Kafka topic is configured to only add data.
- The Kafka key is irrelevant but it is best to use a unique ID such as. a random UUID.
- The Kafka value is the new RDF data to be added.

3.1.3.3 SPARQL templates

The built-in SPARQL template plugin makes it possible to create predefined SPARQL templates that you can use for smart updates to the repository data. All of these operations will behave exactly like any other RDF data.

The plugin is defined with the special predicate `<http://www.ontotext.com/sparql/template>`.

You can create and execute SPARQL templates in Workbench from both the SPARQL editor and from the SPARQL Templates editor.

From the SPARQL editor

Create template

We will use the template from the above *example*. Start by executing the following:

```
INSERT DATA {
    <http://example.com/my-template> <http://www.ontotext.com/sparql/template> '''
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX factory: <http://factory/>
        DELETE {
            ?worker factory:hasSalary ?oldSalary .
        } INSERT {
            ?id factory:updatedOn ?now
        } WHERE {
            ?id rdf:type factory:Factory .
            ?worker factory:worksIn ?id .
            ?worker factory:hasSalary ?oldSalary .
            bind(now() as ?now)
        }
        '''
    }
```

Get template content

```
SELECT ?template {
    <http://example.com/my-template> <http://www.ontotext.com/sparql/template> ?template
}
```

This will return the content of the template, like this:

```
"
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX factory: <http://factory/>
    DELETE {
        ?worker factory:hasSalary ?oldSalary .
    } INSERT {
        ?id factory:updatedOn ?now
    } WHERE {
        ?id rdf:type factory:Factory .
        ?worker factory:worksIn ?id .
        ?worker factory:hasSalary ?oldSalary .
        bind(now() as ?now)
    }
"
```

List defined templates

```
SELECT ?id ?template {
    ?id <http://www.ontotext.com/sparql/template> ?template
}
```

This will list the IDs of the available templates (in our case `http://example.com/my-template`) and their content.

Update template

We can also update the content of the template with the same update operation from earlier:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
INSERT DATA {
    <http://example.com/my-template> <http://www.ontotext.com/sparql/template> '''
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX factory: <http://factory/>
        DELETE {
            ?worker factory:hasSalary ?oldSalary .
        } INSERT {
            ?id factory:updatedOn ?now
        } WHERE {
            ?id rdf:type factory:Factory .
            ?worker factory:worksIn ?id .
            ?worker factory:hasSalary ?oldSalary .
            bind(now() as ?now)
        }
    '''
}
```

Delete template

```
DELETE WHERE {
    <http://example.com/my-template> <http://www.ontotext.com/sparql/template> ?template
}
```

From the SPARQL Templates editor

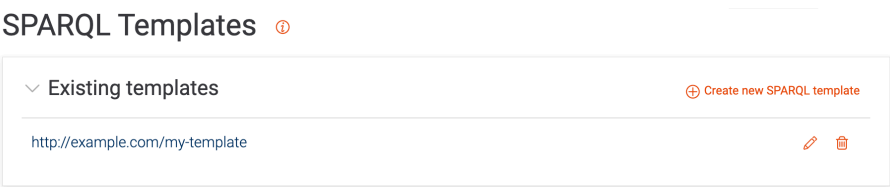
For ease of use, the GraphDB Workbench also offers a separate menu tab where you can define your templates.

1. Go to *Setup* ▸ *SPARQL Templates* ▸ *Create new SPARQL template*. A default example template will open.
2. The template ID is required and must be an IRI. We will use the example from earlier: `http://example.com/my-template`.

   If you enter an invalid IRI, the SPARQL template editor will warn you of it.
3. The template body contains a default template. Replace it with:

   ```
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX factory: <http://factory/>
   DELETE {
       ?worker factory:hasSalary ?oldSalary .
   } INSERT {
       ?id factory:updatedOn ?now
   } WHERE {
       ?id rdf:type factory:Factory .
       ?worker factory:worksIn ?id .
       ?worker factory:hasSalary ?oldSalary .
       bind(now() as ?now)
   }
   ```

   This template can be used for smart updates to the RDF data as shown above.
4. *Save* the template. It will now be visible in the list with created templates where you can also edit or delete it.



SPARQL Template endpoint

In some cases, you may want to execute arbitrary SPARQL updates, storing not the variables but rather the relationship between those variables and the database. An easy way to do that is through the GraphDB REST API SPARQL template endpoint. Let's look at an example.

1. First, we need to import some data with which we will be working.

   Go to *Import* ▸ *User data* ▸ *Import RDF text snippet* and import the following `sample data` describing five fictitious wines:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
```

(continues on next page)

(continued from previous page)

```
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

2. Next, let's create the SPARQL template.

   Go to *Setup* ▸ *SPARQL Templates* ▸ *Create new SPARQL template* and create the following template:

```
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

DELETE {
    ?s wine:hasSugar ?oldValue .
    ?s wine:hasYear ?oldYear
} INSERT {
    ?s wine:hasSugar ?sugar .
    ?s wine:hasYear ?year .
} WHERE {
    ?s ?p ?oldValue .
    ?s ?p1 ?oldYear .
}
```

3. Let's run a SPARQL query against the data. In the SPARQL editor, execute this query:

```
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?s ?p ?o
WHERE {
   BIND(wine:Blanquito as ?s ) .
    ?s ?p ?o .
}
```

   GraphDB will return the following results:

	s	p	o
1	onto:example/wine#Blanquito	rdf:type	onto:example/wine#Wine
2	onto:example/wine#Blanquito	rdf:type	onto:example/wine#WhiteWine
3	onto:example/wine#Blanquito	onto:example/wine#madeFromGrape	onto:example/wine#Chardonnay
4	onto:example/wine#Blanquito	onto:example/wine#hasSugar	"dry"
5	onto:example/wine#Blanquito	onto:example/wine#hasYear	"2012"^^xsd:integer

4. **Example 1**:

   To change the values of the variables for sugar content and year, we will update the data through the REST API endpoint.

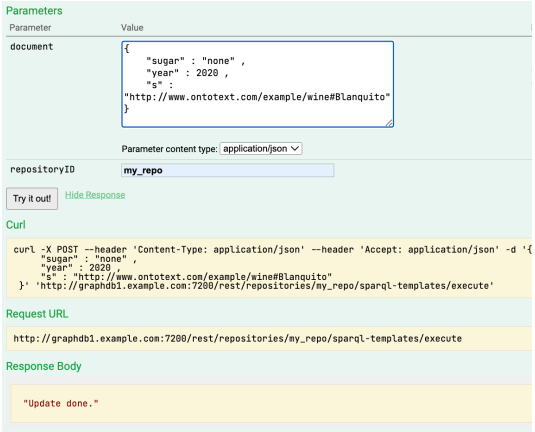
   a. Go to *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *SPARQL Template Controller* ▸ *POST /rest/repositories/{repositoryID}/sparql-templates/execute*.

b. For the *repositoryID* parameter, enter the name of your repository such as “my_repo”.

c. In the *document* field, enter the JSON document:

```
{
    "sugar" : "none" ,
    "year" : 2020 ,
    "s" : "http://www.example.com/demo/wine#Blanquito"
}
```

d. Click *Try it out!*.



e. To see how the data have been updated, let’s execute the SPARQL query from step 3 again:

	s	p	o
1	onto:example/wine#Blanquito	rdf:type	onto:example/wine#Wine
2	onto:example/wine#Blanquito	rdf:type	onto:example/wine#WhiteWine
3	onto:example/wine#Blanquito	onto:example/wine#madeFromGrape	onto:example/wine#Chardonnay
4	onto:example/wine#Blanquito	onto:example/wine#hasSugar	"none"
5	onto:example/wine#Blanquito	onto:example/wine#hasYear	"2020"^^xsd:int

We can see that the objects for the sugar content and year predicates have been updated to “none” and “2020”, respectively.

In this example, we executed a template and added specific values for its variables. Even if we had not specified the type for `2020`, we would get a typed result: `"2020"^^xsd:int`. This is because standard IRIs, numbers, and boolean values are recognized and parsed this way.

5. **Example 2**:

We can also create typed values explicitly by using JSON-LD-like values.

a. We will be using the same SPARQL template as in example 1.

b. Again in *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *SPARQL Template Controller* ▸ *POST /rest/repositories/{repositoryID}/sparql-templates/execute*, send:

```
{
    "sugar" : { "@id" : "custom:iri" } ,

    "year" : { "@value" : "2020" ,
                          "@type" : "http://test.type" } ,

    "s" : "http://www.example.com/demo/wine#Blanquito"
}
```

Most IRIs will be recognized, but some custom ones will not. Here, we are using a special label `@id` so that the value for `sugar` can be parsed as an IRI, since the value `custom:iri` will not be considered an IRI by default.

c. To see how the data have been updated, execute the query from example 1 in the SPARQL editor. The following shows the results:

	s	p	o
1	onto:example/wine#Blanquito	rdf:type	onto:example/wine#Wine
2	onto:example/wine#Blanquito	rdf:type	onto:example/wine#WhiteWine
3	onto:example/wine#Blanquito	onto:example/wine#madeFromGrape	onto:example/wine#Chardonnay
4	onto:example/wine#Blanquito	onto:example/wine#hasSugar	custom:iri
5	onto:example/wine#Blanquito	onto:example/wine#hasYear	"2020"^^<http://test.type>

As shown in the first example, the values will get a type if recognized. If we have a value not in its default type, we can use JSON-LD-like values containing both the `@value` and the `@type`. This example demonstrated this with the `year` variable — the result is `"2020"^^<http://test.type>`.

GraphDB supports SHACL validation ensuring efficient data consistency checking.

3.1.4 SHACL validation

3.1.4.1 What is SHACL validation?

Validation with the W3C standard Shapes Constraint Language (SHACL) standard is a valuable tool for efficient data consistency checking, and is supported by GraphDB via RDF4J's ShaclSail storage and interface layer. It is useful in efforts towards data integration, as well as examining data compliance — for example, requiring that every GeoName URI must start with `https://sws.geonames.org/\d+/`, or that age values must be above 18 years.

SHACL validates RDF graphs against a set of conditions. These conditions are provided as shapes and other constructs expressed in the form of an RDF graph. RDF graphs that are used in this manner are called shapes graphs, and the RDF graphs that are validated against a shapes graph are called data graphs.

A shape is an IRI or a blank node `s` that fulfills at least one of the following conditions in the shapes graph:

- `s` is a SHACL instance of `sh:NodeShape` or `sh:PropertyShape`.
- `s` is subject of a triple that has `sh:targetClass`, `sh:targetNode`, `sh:targetObjectsOf`, or `sh:targetSubjectsOf` as predicate.
- `s` is subject of a triple that has a parameter as predicate.
- `s` is a value of a shape-expecting, non-list-taking parameter such as `sh:node`, or a member of a SHACL list that is a value of a shape-expecting and list-taking parameter such as `sh:or`.

Every GraphDB SHACL repository contains the ShaclSail reserved graph `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph`, where all the data is inserted.

It is also possible to specify your own custom graph via the `sh:shapesGraph` property. See more on this in *Creating and configuring a SHACL repository*.

3.1.4.2 Usage

Creating and configuring a SHACL repository

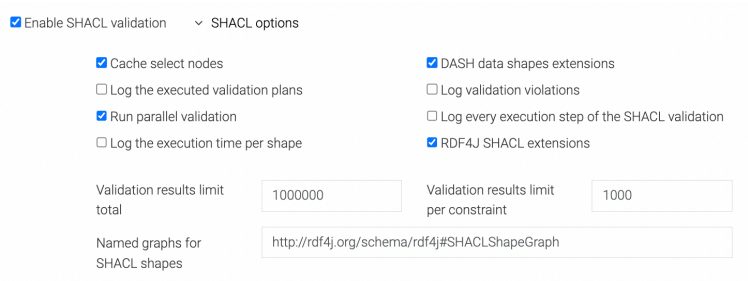
SHACL support in a given repository must be enabled when that repository is created. You cannot modify an already existing repository by enabling the validation afterwards.

When you *create a new GraphDB repository*, *Enable SHACL validation* is a checkbox on the *Create GraphDB repository* screen. Selecting it displays several additional fields that let you customize the use of SHACL:

- *Cache select nodes*: The ShaclSail retrieves a lot of its relevant data by running SPARQL `SELECT` queries against the underlying Sail and against the changes in the transaction. This is usually good for performance, but it's best to disable this cache while validating large amounts of data to consume less memory. Default value is `true`.

- *Log the executed validation plans*: Logs (`INFO`) the executed validation plans using GraphViz DOT format. It's best to disable *Run parallel validation*. Default value is `false`.
- *Run parallel validation*: Runs validation in parallel. May cause deadlock, especially when using NativeStore. Default value is `true`.
- *Log the execution time per shape*: Logs (`INFO`) the execution time per shape. It's best to disable *Run parallel validation* and *Cache select nodes*. Default value is `false`.
- *DASH data shapes extensions*: Activates DASH Data Shapes extensions. DASH Data Shapes Vocabulary is a collection of reusable extensions to SHACL for a wide range of use cases. Currently, this enables support for `dash:hasValueIn`, `dash:AllObjectsTarget`, and `dash:AllSubjectsTargetIt`.
- *Log validation violations*: Logs (`INFO`) a list of violations and the triples that caused the violations (`BETA`). It's best to disable *Run parallel validation*. Default value is `false`.
- *Log every execution step of the SHACL validation*: Logs (`INFO`) every execution step of the SHACL validation. This is fairly costly and should not be used in production. It's best to disable *Run parallel validation*. Default value is `false`.
- *RDF4J SHACL extensions*: Activates RDF4J's SHACL extensions (RSX) that provide additional features. RSX currently contains `rsx:targetShape` which will allow a Shape to be the target for your constraints. For more information about the RSX features, see the RSX section of RDF4J documentation.
- *Named graphs for SHACL shapes*: Sets the named graphs where SHACL shapes can be stored using a comma-delimited list. You can store shapes in one graph and untrusted data in another without the untrusted data overriding the constraints provided by the shapes.

  When using named graphs for SHACL shapes each data graph is evaluated separately by default. For example, if you have `<http://example.org> sh:shapesGraph ex:shaclGraph` and `<http://example2.org> sh:shapesGraph ex:shaclGraph`, the data in the two example graphs will not be joined before being validated. It is not possible to split a single object across multiple graphs in order to validate it with SHACL.



Some of these are used for logging and validation. See more about this in the *Validation logging and report* section.

Loading shapes and data graphs

You can load shapes using all three key methods for loading data into GraphDB: through Workbench, with an `INSERT` query in the SPARQL editor, and through the REST API.

The following shows an example of doing it through Workbench:

1. Go to *Import* ▸ *User data* ▸ *Import RDF text snippet*, and insert the following shape:

```
prefix ex: <http://example.com/ns#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ex:PersonShape
    a sh:NodeShape  ;
```

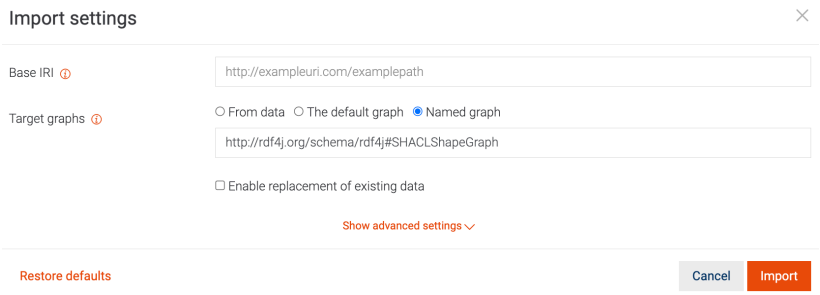
(continues on next page)

(continued from previous page)

```
    sh:targetClass ex:Person ;
    sh:property [
        sh:path ex:age ;
        sh:datatype xsd:integer ;
] .
```

It indicates that entities of the class Person must have a property "age" of the type `xsd:integer`.

Click *Import*. In the dialog box that opens, select *Target graphs* ▸ *Named graph*. Insert the ShaclSail reserved graph `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph` (or a custom named graph specified with the `sh:shapesGraph` property) as shown below:



> **Warning:** There are two reserved graphs when working with SHACL shapes — the default (unnamed) graph and the RDF4J SHACL Shape Graph (`http://rdf4j.org/schema/rdf4j#SHACLShapeGraph`). If you put the SHACL shapes in either of those two graphs, they will always be used for validation when using the SHACL Validator API.

If you want to specify a custom graph to keep the shapes in (for example <http://example.com>) you need to add the link between the data and shapes. In the example below, your data is in `ex:dataGraph` and the shapes graph is `<http://example.com>`:

```
PREFIX ex: <http://example.com/ns#>
PREFIX sh: <http://www.w3.org/ns/shacl#>
INSERT DATA
{
    GRAPH <http://example.com> {
        ex:dataGraph sh:shapesGraph <http://example.com>  .
    }
}
```

2. After the shape has been imported, let's test it with some data:

   a. Again from *Import* ▸ *User data* ▸ *Import RDF text snippet*, insert **correct data**, with as as an integer:

   ```
   prefix ex: <http://example.com/ns#>
   prefix xsd: <http://www.w3.org/2001/XMLSchema#>

   ex:Alice
     rdf:type ex:Person ;
     ex:age 12 ;
   .
   ```

   Leave the *Import settings* as they are and click *Import*. You will see that the data has been imported successfully because it complies with the shape you just inserted.

   b. Now import **incorrect data**, with age as a double:

```
prefix ex: <http://example.com/ns#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>


ex:Alice
  rdf:type ex:Person ;
  ex:age 12.1 ;
.
```

The import will fail, returning a detailed error message that lists all validation violations in both Workbench and the command line.

Deleting shapes and data graphs

There are two ways to delete a SHACL shape: from the GraphDB Workbench and with the RDF4J Java API.

From Workbench

1. Go to the SPARQL Editor in Workbench.
2. Clear the RDF4J graph for storing shapes by running the following update query:

```
CLEAR GRAPH <http://rdf4j.org/schema/rdf4j#SHACLShapeGraph>
```

Note: Keep in mind that the *Clear Repository* option in the *Explore* ▸ *Graphs overview* tab would not delete the shape graph, as it removes all data from the repository, but not SHACL shapes.

With the RDF4J Java API

Use the following code snippet:

```
HTTPRepository repository = new HTTPRepository("http://address:port/", "repositoryname");
try (RepositoryConnection connection = repository.getConnection()) {
        connection.begin();
        connection.clear(RDF4J.SHACL_SHAPE_GRAPH);
    connection.commit();
}
```

Bulk validation on an existing repository with external sources

You can also use the REST API to run SHACL validation on an existing repository regardless of the initial repository configuration by providing SHACL shapes as a file, as a string of text, or with a pointer to shapes stored in an existing repository. Examples here use `curl`.

Bulk validation with shapes stored in a file

```
POST /rest/repositories/{repositoryID}/validate/file
```

Example code snippet (`shapes.ttl` refers to a file local to where `curl` is called):

```
curl -X POST --header 'Accept: text/turtle'\
    'http://localhost:7200/rest/repositories/dataRepo/validate/file'\
    -F 'file=@shapes.ttl;type=text/turtle'
```

Bulk validation with shapes provided text

```
POST /rest/repositories/{repositoryID}/validate/text
```

Example code snippet:

```
curl -X POST --header 'Content-Type: text/turtle' --header 'Accept: text/turtle'\
    'http://localhost:7200/rest/repositories/dataRepo/validate/text'\
    --data-raw '@prefix ex: <http://example.com/ns#> .
                @prefix sh: <http://www.w3.org/ns/shacl#> .
                @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
                ex:PersonShape
                    a sh:NodeShape  ;
                    sh:targetClass ex:Person ;
                    sh:property [
                        sh:path ex:age ;
                        sh:datatype xsd:integer ;
                ] .'
```

Bulk validation with shapes stored in another repository

```
POST /rest/repositories/{dataRepositoryID}/validate/repository/{shapesRepositoryID}
```

```
curl -X POST --header 'Accept: text/turtle'\
    'http://localhost:7200/rest/repositories/dataRepo/validate/repository/shapesRepo'
```

For this example, *import the following shape as an RDF text snippet* in the `dataRepo` repository:

```
prefix ex: <http://example.com/ns#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ex:Alice
  rdf:type ex:Person ;
  ex:age 12.1 .

ex:Bob
  rdf:type ex:Person ;
  ex:age 12 .
```

Use the following shape for the `shapesRepo` against which data in the `dataRepo` repository will be validated:

```
prefix ex: <http://example.com/ns#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
```

(continues on next page)

(continued from previous page)

```
ex:PersonShape
    a sh:NodeShape  ;
    sh:targetClass ex:Person ;
    sh:property [
        sh:path ex:age ;
        sh:datatype xsd:integer
    ] .
```

Bulk validation with a named graph in another repository

When validating data against another existing repository (referred to as a shapes repository), GraphDB will use the default graph and the SHACL default graph of that repository to validate all named graphs in the repository that you are validating. To validate your data using named graphs from the shapes repository, you need to define the mapping between data graphs to validate from the data repository, and shapes graphs from the shapes repository to validate against.

Here is example data for the `dataRepo` repository for such a scenario:

```
prefix ex: <http://example.com/ns#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ex:doNotValidate {
  ex:Alice a ex:Person;
    ex:age 12.1 .
}

{
  ex:Bob a ex:Person;
    ex:age 12 .

  ex:Aerith a ex:Person;
    ex:age 12.1 .
}

ex:validate {
  ex:Steve a ex:Person;
    ex:age 12.5 .
}
```

Use the following shape for the `shapesRepo` that will be used to validate the `dataRepo` data:

```
prefix ex: <http://example.com/ns#>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

ex:shapesGraph {
  ex:PersonShape a sh:NodeShape;
    sh:targetClass ex:Person;
    sh:property _:node2 .

  _:node2 sh:path ex:age;
    sh:datatype xsd:integer .
}

{
  ex:validate sh:shapesGraph ex:shapesGraph .
}
```

In this example, the data graph `ex:validate` will be validated against the shapes graph `ex:shapesGraph`. The

mapping is defined in the default graph of the shapes repository. It will find a violation because an age value must be an integer. However, neither the named graph `ex:doNotValidate` containing suspicious data nor the default graph which contains both normal and suspicious data will be validated, as these are not mentioned in the mapping provided in the shapes repository.

Union of data graphs when validating the results

> **Warning:** This feature is experimental and can hurt performance.

You can make a union of the data graphs when validating the results.

Here is an example, where:

- the triple `ex:Alice rdf:type ex:Person.` is in the named graph `ex:dataGraph`
- the triple `ex:Alice ex:age 12.1` is in the named graph `ex:dataGraph2`

```
PREFIX ex: <http://example.com/ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
INSERT DATA {
    GRAPH ex:dataGraph{
        ex:Alice rdf:type ex:Person.
    }
    GRAPH ex:dataGraph2{
        ex:Alice ex:age 12.1
    }
}
```

This can be done by inserting the following shape in the shapes repository as TriG.

```
@prefix ex: <http://example.com/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf4j: <http://rdf4j.org/schema/rdf4j#> .
@prefix rsx: <http://rdf4j.org/shacl-extensions#> .
<http://example.com> {
  ex:PersonShape a sh:NodeShape;
    sh:targetClass ex:Person;
    sh:property [
        sh:path ex:age ;
        sh:datatype xsd:integer ;
      ] .
    [
      a rsx:DataAndShapesGraphLink;
      rsx:shapesGraph <http://example.com>;
      rsx:dataGraph ex:dataGraph, ex:dataGraph2;
    ]
}
```

When you run the validation against the validation endpoint you will get both graphs in `rsx:dataGraph` `/rest/repositories/{repositoryID}/validate/repository/{shapesRepositoryID}`

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rsx: <http://rdf4j.org/shacl-extensions#> .
@prefix dash: <http://datashapes.org/dash#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

(continues on next page)

(continued from previous page)

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf4j: <http://rdf4j.org/schema/rdf4j#> .

[] a sh:ValidationReport;
  sh:conforms false;
  rdf4j:truncated false;
  sh:result [ a sh:ValidationResult;
      sh:focusNode <http://example.com/ns#Alice>;
      rsx:dataGraph <http://example.com/ns#dataGraph>, <http://example.com/ns#dataGraph2>;
      rsx:shapesGraph <http://example.com>;
      sh:value 12.1;
      sh:resultPath <http://example.com/ns#age>;
      sh:sourceConstraintComponent sh:DatatypeConstraintComponent;
      sh:resultSeverity sh:Violation;
      sh:sourceShape [ a sh:PropertyShape;
          sh:path <http://example.com/ns#age>;
          sh:datatype xsd:integer
        ]
    ]
```

Updating shapes and data graphs

To update a shape graph:

1. Go to the SPARQL Editor in Workbench.
2. Clear the RDF4J graph for storing shapes by running the following update query:

```
CLEAR GRAPH <http://rdf4j.org/schema/rdf4j#SHACLShapeGraph>
```

3. Load the updated shape graph following the instructions in *Loading shapes and data graphs*.

Note: As shape graphs are stored separately from data, importing a new shape graph by enabling the *Enable replacement of existing data box* option in the *Import settings* dialog box would not work. This is why the above steps are necessary.

Viewing shapes and data graphs

Currently, shape graphs cannot be accessed with SPARQL inside GraphDB because they are not part of the data. You can view the graph by using the RDF4J client to connect to the GraphDB repository. The following Java code snippet will return all statements inside the shape graph:

```
HTTPRepository repository = new HTTPRepository("http://address:port/", "repositoryname");
try (RepositoryConnection connection = repository.getConnection()) {
    Model statementsCollector = new LinkedHashModel(connection.getStatements(null, null,
↪null, RDF4J.SHACL_SHAPE_GRAPH)
        .stream()
        .collect(Collectors.toList()));
}
```

You can also view the data by navigating to *Explore* ▸ *Graphs overview* and clicking on the *Export graph* button for the graph with the SHACL shape.

Note: Since SHACL shapes are not part of the data, they are not part of the export options when using the *Export repository* button.

SPARQL capabilities in SHACL shapes

GraphDB supports SPARQL-based constraint components as well as SPARQL-based targets.

SHACL-SPARQL has to be written judiciously for performance reasons. Because `SPARQLConstraint` is executed for each SHACL target, you want to make the targets as few as possible for performance reasons. In practice, this means that the actual constraints can be located within a `sh:SPARQLTarget`, and the `sh:SPARQLConstraint` is mostly used for populating the answers.

The following example has two SPARQL requests: a `SPARQLTarget` that finds one violation and a `SPARQLConstraint` check to clear up the message and value. If using `sh:targetClass` instead of a SPARQL target, GraphDB would execute a SPARQL request for each instance of the `schema:Person` class, impacting performance severely on a dataset of realistic sizes.

```
@prefix dash:   <http://datashapes.org/dash#> .
@prefix schema: <http://schema.org/> .
@prefix sh:     <http://www.w3.org/ns/shacl#> .

schema:
  sh:declare [
    sh:prefix "schema" ;
    sh:namespace "http://schema.org/" ;
  ] .

schema:PersonShape
    a sh:NodeShape ;
    sh:target [
        a sh:SPARQLTarget;
        sh:prefixes schema: ;
        sh:select """
            select ?this {
                ?this a schema:Person ;
                      schema:age ?age .
                FILTER (?age < 18)}"""];
  sh:sparql [
    a sh:SPARQLConstraint;
    sh:prefixes schema: ;
    sh:message "This person is too young to vote!";
    sh:select """
      select $this ?value {
        $this schema:age ?value .
      }"""
    ].
```

```
PREFIX ex:     <http://example.org/ns#>
PREFIX schema: <http://schema.org/>
INSERT DATA {
    ex:Aerith
        a schema:Person ;
        schema:age 2 ;
        schema:eligibleToVote false .
    ex:Bob
        a schema:Person ;
        schema:age 21 ;
        schema:eligibleToVote true .
    ex:Alice
```

(continues on next page)

(continued from previous page)

```
        a schema:Person ;
        schema:age 16 ;
        schema:eligibleToVote true .
}
```

Note: At the moment, `sh:message` does not populate values from `sh:SPARQLConstraint`.

3.1.4.3 Validation logging and report

ShaclSail validates the data changes on `commit()`. In case of a violation, it will throw an exception that contains a validation report where you can find details about the noncompliance of your data. The exception will be shown in Workbench if it was caused by an update executed in the same Workbench window.

You may also enable ShaclSail logging to get additional validation information in the log files. To enable logging, check one of the three logging options when creating the SHACL repository:

- *Log the executed validation plans* — check RDF4J's ShaclSail documentation on logging validation plans.
- *Log validation violations* — check RDF4J's ShaclSail documentation on logging validation violations.
- *Log every execution step of the SHACL validation* — check RDF4J's ShaclSail documentation on logging validation execution.

All three will log as `INFO` entries and appear in the `main-[yyyy-mm-dd].log` file in the `logs` directory of your GraphDB installation.

3.1.4.4 Supported SHACL features

Feature	Description
`sh:targetClass`	Specifies a target class. Each value of `sh:targetClass` in a shape is an IRI.
`sh:targetNode`	Specifies a node target. Each value of `sh:targetNode` in a shape is either an IRI or a literal.
`sh:targetSubjectsOf`	Specifies a subjects-of target in a shape. The values are IRIs.
`sh:targetObjectsOf`	Specifies an objects-of target in a shape. The values are IRIs.
`sh:path`	Points at the IRI of the property that is being restricted. Alternative, it may point at a path expression, which would allow you to constrain values that may be several "hops" away from the starting point.
`sh:inversePath`	An inverse path is a blank node that is the subject of exactly one triple in a graph. This triple has `sh:inversePath` as predicate, and the object is a well-formed SHACL property path.
`sh:property`	Specifies that each value node has a given property shape.
`sh:or`	Specifies the condition that each value node conforms to at least one of the provided shapes.
`sh:and`	Specifies the condition that each value node conforms to all provided shapes. This is comparable to conjunction and the logical "and" operator.
`sh:not`	Specifies the condition that each value node cannot conform to a given shape. This is comparable to negation and the logical "not" operator.
`sh:minCount`	Specifies the minimum number of value nodes that satisfy the condition. If the minimum cardinality value is 0 then this constraint is always satisfied and so may be omitted.
`sh:maxCount`	Specifies the maximum number of value nodes that satisfy the condition.
`sh:minLength`	Specifies the minimum string length of each value node that satisfies the condition. This can be applied to any literals and IRIs, but not to blank nodes.
`sh:maxLength`	Specifies the maximum string length of each value node that satisfies the condition. This can be applied to any literals and IRIs, but not to blank nodes.

Continued on next page

Table 2 – continued from previous page

Feature	Description
`sh:pattern`	Specifies a regular expression that each value node matches to satisfy the condition.
`sh:flags`	An optional string of flags, interpreted as described in the W3C SPARQL Recommendation at SPARQL 1.1 REGEX. The values of `sh:flags` in a shape are literals with datatype `xsd:string`.
`sh:nodeKind`	Specifies a condition to be satisfied by the RDF node kind of each value node.
`sh:languageIn`	Specifies that the allowed language tags for each value node are limited by a given list of language tags.
`sh:datatype`	Specifies a condition to be satisfied with regards to the datatype of each value node.
`sh:class`	Specifies that each value node is a SHACL instance of a given type.
`sh:in`	Specifies the condition that each value node is a member of a provided SHACL list.
`sh:message`	Specifies a human-readable message that can be associated with a shape. The `sh:resultMessage` property provides that message for a particular shape violation, and it's best to base it on the value of the `sh:message` property for a particular shape. At the moment, `sh:message` does not populate its content with variables from the report.
`sh:severity`	Specifies a severity value for the `sh:severity` property in the shapes graph. Each value of `sh:severity` in a shape is an IRI. SHACL describes three types of severity levels: `sh:info`, `sh:warning`, and `sh:violation`.
`sh:uniqueLang`	Can be set to true to specify that no pair of value nodes may use the same language tag.
`sh:minInclusive`	Specifies the minimum inclusive value. The values of `sh:minInclusive` in a shape are literals. A shape has at most one value for `sh:minInclusive`.
`sh:maxInclusive`	Specifies the maximum inclusive value. The values of `sh:maxInclusive` in a shape are literals. A shape has at most one value for `sh:maxInclusive`.
`sh:minExclusive`	Specifies the minimum exclusive value. The values of `sh:minExclusive` in a shape are literals. A shape has at most one value for `sh:minExclusive`.
`sh:maxExclusive`	Specifies the maximum exclusive value. The values of `sh:maxExclusive` in a shape are literals. A shape has at most one value for `sh:maxExclusive`.
`sh:deactivated`	A shape that has the value `true` for the property `sh:deactivated` is called deactivated. The value of `sh:deactivated` in a shape must be either `true` or `false`.
`sh:hasValue`	Specifies the condition that at least one value node is equal to the given RDF term.
`sh:shapesGraph`	Sets the named graphs where SHACL shapes can be stored. Comma-delimited list.
`dash:hasValueIn`	Can be used to state that at least one value node must be a member of a provided SHACL list. This constraint component only makes sense for property shapes. It takes a list argument similar to `sh:in` but is "open" like `sh:hasValue` since it allows values outside of the list.
`sh:target`	For use with DASH targets.
`rsx:targetShape`	Part of RDF4J's SHACL extensions (RSX) that allows a shape to be the target for your constraints. For more information about the RSX features, see the RSX section.

Implicit `sh:targetClass` is supported for nodes that are `rdfs:Class` and either of `sh:PropertyShape` or `sh:NodeShape`. For all nodes that are equivalent to `owl:Thing` in an environment with a reasoner, you can enable validation by setting `setUndefinedTargetValidatesAllSubjects(true)`.

`sh:or` is limited to statement-based restrictions such as `sh:datatype`, or aggregate-based restrictions such as `sh:minCount`, but not both at the same time.

> **Warning: The above description of sh:path is correct when all sh:paths are supported.** Currently, `sh:path` is limited to predicate paths, sequence paths, alternative paths, and inverse paths. Support for zero or more, one or more, or zero or one paths will be added in the near future.

The GraphDB sequences plugin provides transactional sequences for GraphDB. A sequence is a long counter that can be atomically incremented in a transaction to provide incremental IDs.

3.1.5 Generating sequential IDs with the Sequences plugin

3.1.5.1 What the plugin does

The Sequences plugin provides transactional sequences for GraphDB. A sequence is a long counter that can be atomically incremented in a transaction to provide incremental IDs.

3.1.5.2 Usage

The plugin supports multiple concurrent sequences where each sequence is identified by an IRI chosen by the user.

Creating a sequence

Choose an IRI such as `http://example.com/my/seq1` for your sequence. Insert the following triple to create a sequence whose next value will be 1:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

INSERT DATA {
    my:seq1 seq:create []
}
```

You can also create a sequence by providing the starting value — for example, to create a sequence whose next value will be `10`:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

INSERT DATA {
    my:seq1 seq:create 10
}
```

When using the GraphDB cluster, you might get the following exception if the repository existed before registering the plugin: `Update would affect a disabled plugin: sequences`. You can activate the plugin with the following update request:

```
INSERT DATA { [] <http://www.ontotext.com/owlim/system#startplugin> "sequences".}
```

Using a sequence

Processing sequence values on the client

In this scenario, new and current sequence values can be retrieved on the client where they can be used to generate new data that can be added to GraphDB in the same transaction.

Note: Using the below examples will not work inside the GraphDB Workbench because they need to be executed in one single transaction. If run one by one, they would be performed in separate transactions. For more information on how to execute them in one transaction, see *Using sequence values only on the server*.

To use any sequence, you must first start a transaction and then prepare the sequences for use by executing the following update:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>

INSERT DATA {
    [] seq:prepare []
}
```

Then you can request new values from any sequence by running a query like this (for the sequence `http://example.com/my/seq1`):

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

SELECT ?next {
    my:seq1 seq:nextValue ?next
}
```

To query the last new value without incrementing the counter, you can use a query like this:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

SELECT ?current {
    my:seq1 seq:currentValue ?current
}
```

Use the obtained values to construct IRIs, assign IDs, or for any other use case.

Using sequence values only on the server

In this scenario, new and current sequence values are available only within the execution context of a SPARQL `INSERT` update. You can generate new data using the sequence values and add that data to GraphDB with the same `INSERT` statement.

The following example prepares the sequences for use and inserts some new data using the sequence `http://example.com/my/seq1` where the subject of the newly inserted data is created from a value obtained from the sequence.

The example will work both in:

- the GraphDB cluster, because new sequence values do not need to be exposed to the client
- the GraphDB Workbench, because it performs everything in a single transaction by separating individual operations using a semicolon

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

# Prepares sequences for use
INSERT DATA {
    [] seq:prepare []
};

# Obtains a new value from the sequence and creates an IRI based on it,
# then inserts new triples using that IRI
INSERT {
    ?subject rdfs:label "This is my new document" ;
            a my:Type1
} WHERE {
    my:seq1 seq:nextValue ?next
    BIND(IRI(CONCAT("http://example.com/my-data/test/", STR(?next))) as ?subject)
};
```

(continues on next page)

(continued from previous page)

```
# Retrieves the last obtained value, recreates the same IRI,
# and adds more data using the same IRI
INSERT {
    ?subject rdfs:comment ?comment ;
} WHERE {
    my:seq1 seq:currentValue ?current
    BIND(IRI(CONCAT("http://example.com/my-data/test/", STR(?current))) as ?subject)
    BIND(CONCAT("The document ID is ", STR(?current)) as ?comment)
}
```

After that, commit the transaction.

Dropping a sequence

Dropping a sequence is similar to creating it. For example, to drop the sequence `http://example.com/my/seq1`, execute this:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

INSERT DATA {
    my:seq1 seq:drop []
}
```

Resetting a sequence

In some cases, you might want to reset an existing sequence such that its next value will be a different number. Resetting is equivalent to dropping and recreating the sequence.

To reset a sequence such that its next value will be `1`, execute this update:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

INSERT DATA {
    my:seq1 seq:reset []
}
```

You can also reset a sequence by providing the starting value. For example, to reset a sequence such that its next value will be `10`, execute:

```
PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
PREFIX my: <http://example.com/my/>

INSERT DATA {
    my:seq1 seq:reset 10
}
```

3.1.6 Loading via HTTP with curl

Using `curl` lets you script this call in an application. See also the *Help ▸ REST API* view of the GraphDB Workbench where you will find a complete reference of all REST APIs and be able to run API calls directly from the browser.

In addition to this, the RDF4J API is also available.

Most data import queries can either take the following set of attributes as an argument or return them as a response.

- `fileNames` (string list): A list of the files to import.
- `importSettings` (JSON object): Import settings.
  - `baseURI` (string): Base URI for the files to be imported.
  - `context` (string): Context for the files to be imported.
  - `data` (string): Inline data.
  - `forceSerial` (boolean): Force use of the serial statements pipeline.
  - `name` (string): Filename.
  - `status` (string): Status of an import — pending, importing, done, error, none, or interrupting.
  - `timestamp` (integer): When the import was started.
  - `type` (string): The type of the import.
  - `replaceGraphs` (string list): A list of graphs that you want to be completely replaced by the import.
  - `parserSettings` (JSON object): Parser settings.
    * `failOnUnknownDataTypes` (boolean): Fail parsing if datatypes are not recognized.
    * `failOnUnknownLanguageTags` (boolean): Fail parsing if languages are not recognized. Valid language tags are defined in the IANA Language Subtag Registry. Since its structure (described in BCP47 sec 3.1) is not easy to read, the script named iana-lang-tags.pl gets the registry, parses it, and writes it to a tab-delimited file. The result is saved as a Google Sheet named `iana-lang-tags`. You can also use custom tags and subtags by using the prefix `x-`.
    * `normalizeDataTypeValues` (boolean): Normalize recognized datatype values. For example, `"01"^^xsd:integer` will be converted to `"1"^^xsd:integer`.
    * `normalizeLanguageTags` (boolean): Normalize recognized language tags, as described in RFC 5646, section 2.1.1.
    * `preserveBNodeIds` (boolean): Use blank node IDs found in the file instead of assigning them.
    * `stopOnError` (boolean): Stop on error. If `false`, the error will be logged and parsing will continue.
    * `verifyDataTypeValues` (boolean): Verify values of recognized datatypes and stop on error. This applies to dates (such as `"2023-02-31"^^xsd:date`), numbers (`"foo"^^xsd:decimal`), decimal digits (`"3.14"^^xsd:integer`), and range errors (`"-1"^^xsd:positiveInteger`, `"1234567890123456"^^xsd:int`). It does not apply to custom datatypes or datatypes derived by restriction.
    * `verifyLanguageTags` (boolean): Verify language based on a given set of definitions for valid languages.
    * `contextLink` (string): Provide context for importing Flattened and Compacted JSON-LD files or NDJSON-LD files.

Cancel server file import operation

```
DELETE /rest/repositories/<repo_id>/import/server
```

Example:

```
curl -X DELETE <base_url>/rest/repositories/<repo-id>/import/server?name=<encoded_filepath>
```

Get server files available for import

```
GET /rest/repositories/<repo_id>/import/server
```

Example:

```
curl <base_url>/rest/repositories/<repo_id>/import/server
```

Import a server file into the repository or a named graph inside the repository

```
POST /rest/repositories/<repo_id>/import/server
```

Example:

```
curl -X POST --header 'Content-Type: application/json' -d '{
  "fileNames": [
    "<data_url>",
    "<data_url>"
  ]
}' <base_url>/rest/repositories/<repo_id>/import/server
```

Tip: Common parameters:

`<base_url>`: The URL host and path leading to the deployed GraphDB Workbench webapp

`<repo_id>`: The ID string that identifies the repository to use

`<encoded_filepath>`: Encoded filepath leading to a server file that is in the process of being imported

With the API, you can add a file of triples to a named graph specified on the command line. It's important to remember that RDF4J calls the fourth part of a quad (where the graph name is stored) the context.

Example:

```
  curl -X POST --header 'Content-Type: text/turtle' \
--data-binary @<data_url> \  '<base_url>/repositories/<repo_id>/statements?context=<named_graph>
```

Warning: The graph name must include <> and the whole name must be URL encoded. In the example above, *<named_graph>* is the name of the graph, including the angle brackets, which must also be encoded.

3.2 Exploring your data

GraphDB offers a wide variety of tools to let you get the most out of the data stored in your repositories. These tools build on support of multiple standards to let you mix and match different ways of exploring the data used by your applications.

- *SPARQL queries*: Using the GraphDB Workbench SPARQL editor to enter, edit, save, share, and manage queries.
- *Ranking results* Sorting query results using an algorithm that identifies the more important or more popular entities in the repository by examining their interconnectedness.
- *Graph path search*: Using GraphDB's built-in implementations of classic graph search algorithms such as shortest path and all paths.

- *Full-text search*: Searching natural language content in a repository's text. These searches use popular full-text search operators that let you perform searches such as "find all documents that have the words 'graph' and 'intelligent' within 5 words of each other".
- *Semantic similarity searches*: Searching for terms and documents that are semantically similar to each other. This takes advantage of machine learning vector space encodings to cluster documents and to find search results with terms that are close in meaning without being the same words as the search terms.
- *Querying GPT models*: Using GraphDB's extensions to the SPARQL query language that let you send natural language queries to Large Language Models.
- *Talk to Your Graph*: Building a chatbot that uses Chat Completions-compatible LLMs in the background so your users can query your GraphDB repositories with conversational natural language input.
- *Geospatial extensions* and *GeoSPARQL support*: Using GraphDB's support for the W3C's RDF model of the WGS 84 geospatial data standard and its support for the Open Geospatial Consortium's SPARQL extension. These let you store and query geospatial data — for example, to query which coffee shops are within 10km of a theater.
- *Data history and versioning*: Storing transaction data that lets you access past states of your database.
- *SQL access over JDBC*: Providing table views of GraphDB data to tools that typically query relational databases.
- *SPARQL Federation*: Executing distributed queries over multiple SPARQL endpoints.
- *Visualize and explore*: Generating interactive diagrams to explore repository data models and instance data.
- *Exporting data*: Exporting data from repositories, graphs, query results, and resources using Workbench or an HTTP interface with `curl`.
- *JavaScript functions*: Defining your own functions in JavaScript to call from your SPARQL queries.
- *The SPARQL-MM multimedia extension*: Using and querying the W3C extension to RDF for audio and video data.

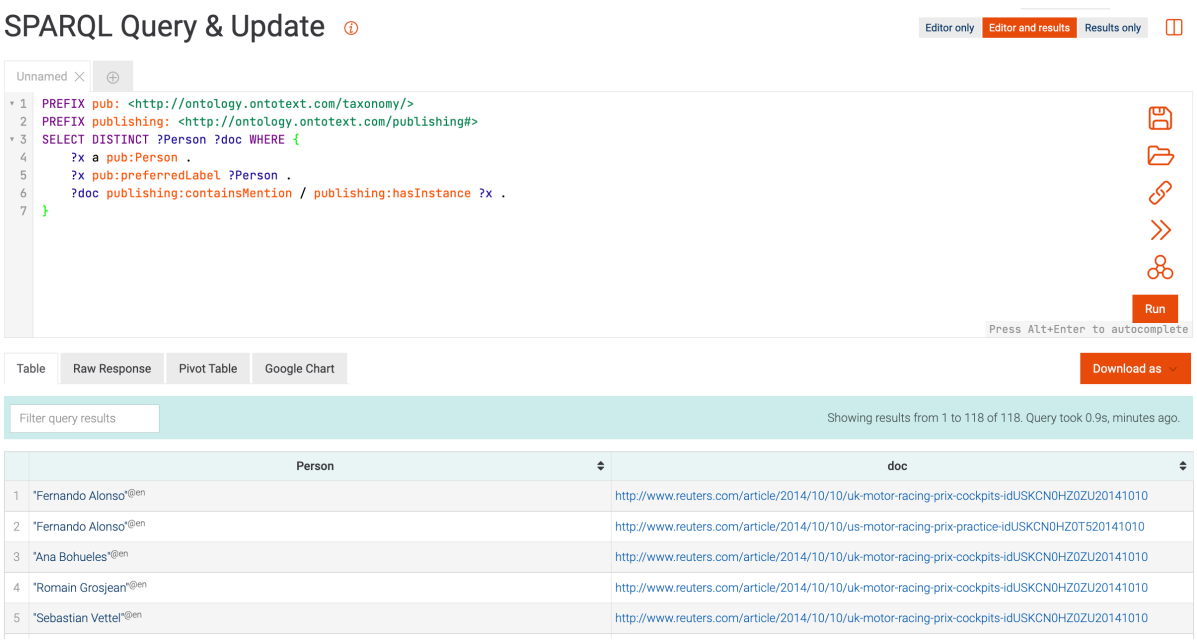
3.3 SPARQL queries

To manage and query your data, go to the *SPARQL* menu tab. The SPARQL view integrates the YASGUI query editor plus some additional features that are described below.

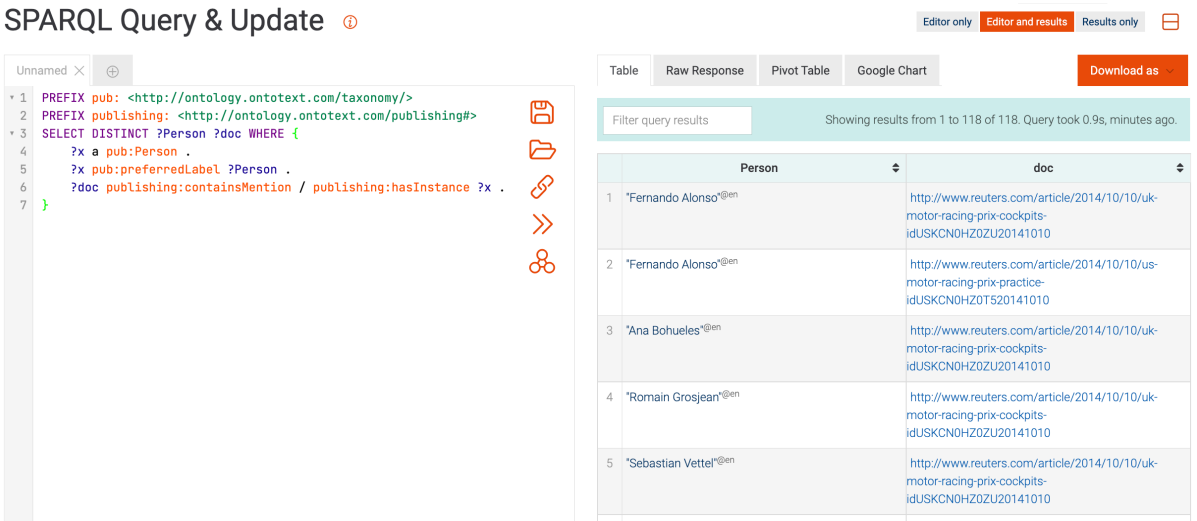
Tip: SPARQL is a SQL-like query language for RDF graph databases with the following request types:

- `SELECT` — returns tabular results
- `CONSTRUCT` — creates a new RDF graph based on query results
- `ASK` — returns "YES" if the query has a solution, otherwise "NO"
- `DESCRIBE` — returns RDF data about a resource; useful when you do not know the RDF data structure in the data source
- `INSERT` — inserts triples into a graph
- `DELETE` — deletes triples from a graph

The SPARQL editor offers two display modes — horizontal and vertical.



Use the vertical mode switch in the upper right ◫ to show the editor and the results next to each other, which is particularly useful on wide screen. Click the switch again to return to horizontal mode.



Both in horizontal and vertical mode, you can also hide the editor or the results to focus on query editing or result viewing. Click the buttons *Editor only*, *Editor and results*, or *Results only* (to the left of the vertical/horizontal mode switch in the upper right) to control which of these is displayed.

Manage your data by writing queries in the text area. It offers syntax highlighting and namespace autocompletion for easy reading and writing.

> **Tip:** To manage the built-in list of declared namespaces, go to *Setup* ▸ *Namespaces*.

Include or exclude inferred statements in the results by clicking the >> icon. When inferred statements are included, both elements of the arrow icon are a solid line (*ON*), otherwise the left element is a solid line and the right one is a dotted line. (*OFF*).

Enable or disable the expanding of the results over `owl:sameAs` by clicking the last icon above the *Run* button. Similarly to the one above it, the setting is *ON* when all its three circles are a solid line, and *OFF* when two of them are a dotted one.

Execute the query by clicking the *Run* button or use `Ctrl/Cmd + Enter`.

Tip: You can find other useful shortcuts in the *keyboard shortcuts* link in the lower right corner of the SPARQL editor.

The results can be viewed in different formats corresponding to the type of the query. By default, they are displayed as a table. Other options are *Raw response*, *Pivot table* and *Google Charts*. You can sort the results by column values and filter them by table values. The query results header will display the total number of results and the time it took to execute the query.

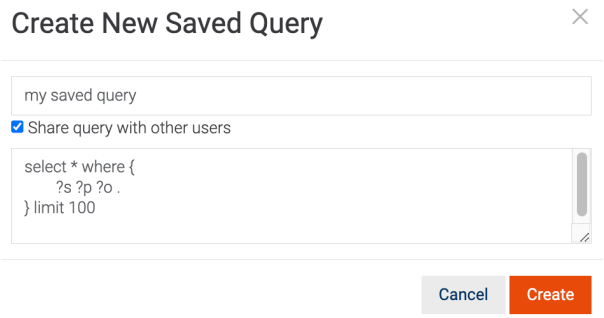
Note: The total number of results is obtained by an async request with a `default-graph-uri` parameter and the value `http://www.ontotext.com/count`.

Navigate through all results by using pagination (SPARQL view can only show a limited number of results at a time). Each page executes the query again with query limit and offset values for `SELECT` queries. For graph queries (`CONSTRUCT` and `DESCRIBE`), all results are fetched by the server and only the page of interest is gathered from the results iterator and sent to the client.

The query results are limited to 1,000, since your browser cannot handle an infinite number of results. Obtain all results by using *Download As* and select the required format for the data. Choices include JSON, XML, CSV, TSV and Binary RDF for `SELECT` queries and all *supported RDF formats* for `CONSTRUCT` and `DESCRIBE` query results.

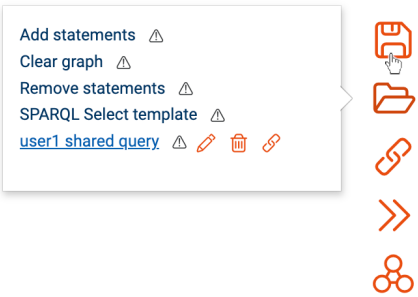
3.3.1 Save and share queries

Use the editor's tabs to keep several queries opened while working with GraphDB. Save a query on the server with the *Create saved query* icon .



When security is ON in the *Setup* ▸ *Users and Access* menu, the system distinguishes between different users. The user can choose whether to share a query with others. A shared query can only be edited by its owner.

The *Show saved queries* icon provides access to your queries and the default and shared ones.

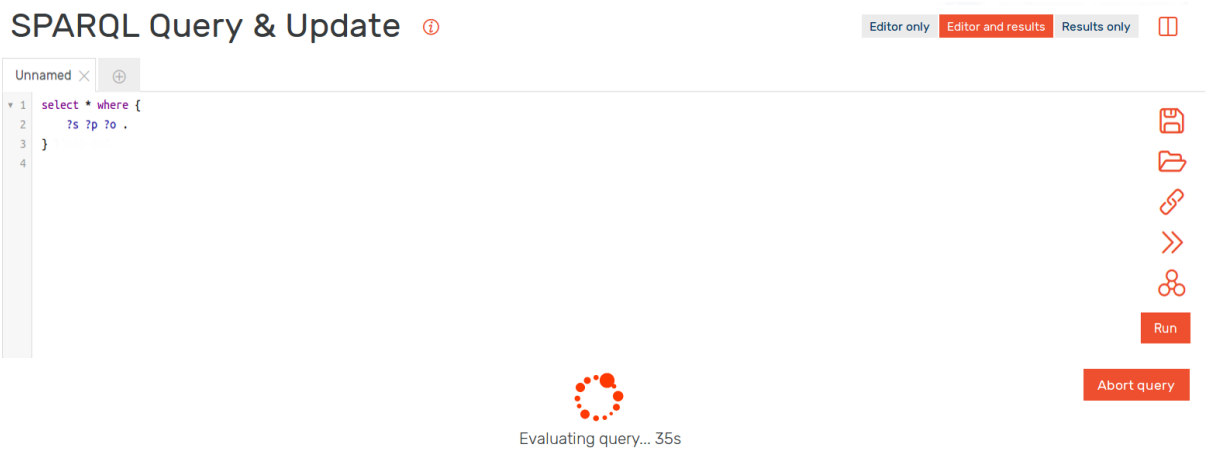


Copy your query as a URL by clicking the *Get URL to current query* icon. This link will also ask the user loading it whether they want GraphDB to automatically update the active repository to the same one as the saved query.

When Free access is ON, the Free Access user will only see shared queries and will not be able to save new queries.

3.3.2 Interrupt queries

You can use the *Abort query* button in the SPARQL editor to manually interrupt any query.



3.4 SPARQL plugins

GraphDB plugins are external libraries that provide additional features. GraphDB can delegate part of its query and update processing tasks to these plugins to trigger the execution of change tracking, entity ranking, graph path search, and additional tasks that happen efficiently because the *GraphDB Plugin API* provides low-level access to GraphDB repository data. The creation of new plugins using this API can provide new features to GraphDB without requiring an upgrade. Below is a list of plugins that are available with this release and information about how to enable or disable them.

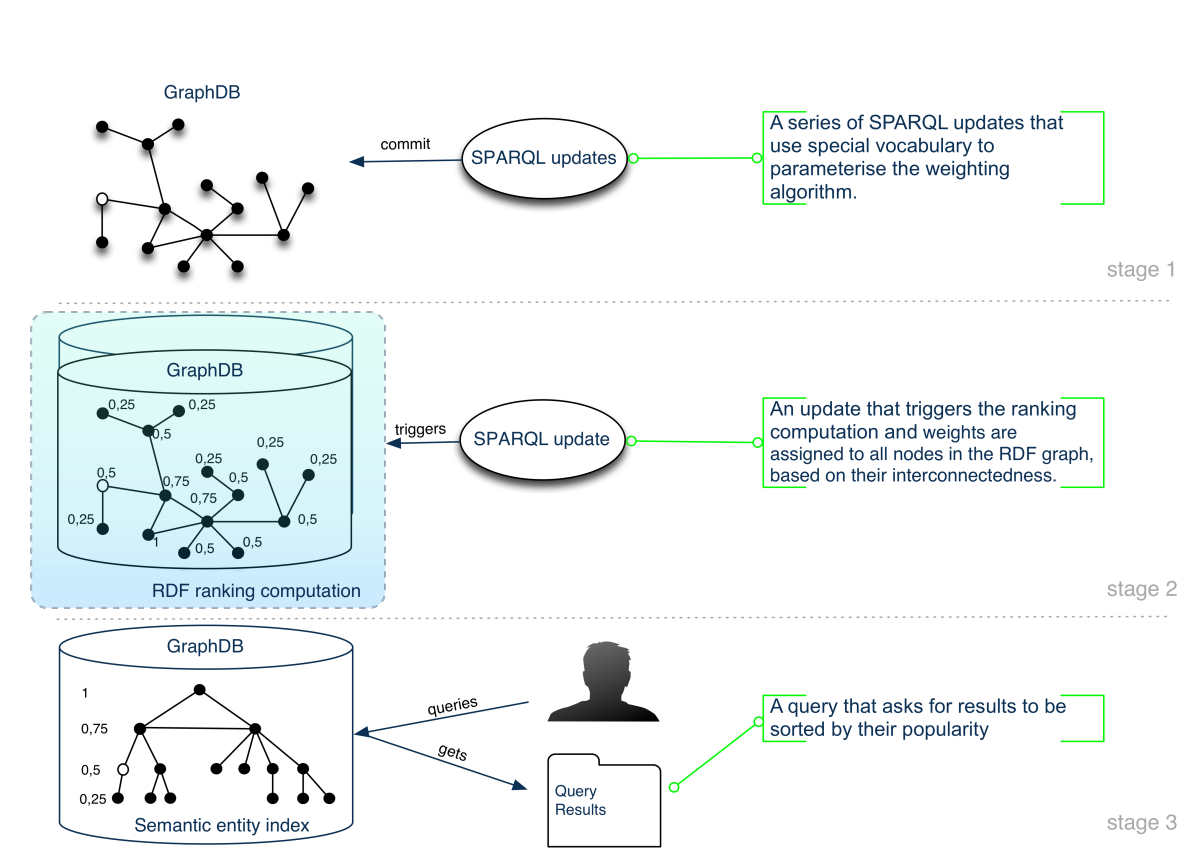
3.4.1 Ranking results

3.4.1.1 Ranking entities by interconnectedness with the RDF Rank Plugin

RDF Rank is an algorithm that identifies the more important or more popular entities in a repository by examining their interconnectedness. The popularity of entities can then be used to sort query results much like internet search engine algorithms such as Google's PageRank.

Note: RDF Rank Plugin and PageRank use different algorithms. Thus, there are differences between the two algorithms when computing the weight of the nodes.

The RDF Rank component computes a numerical weighting for all nodes in the RDF graph stored in the repository, including URIs, blank nodes, literals, and RDF-star embedded triples. The weights are floating point numbers with values between 0 and 1 that can be interpreted as a measure of a node's relevance or popularity.



Since the values range from 0 to 1, the weights can be used to sort a result set. The lexicographical order works fine even if the rank literals are interpreted as plain strings.

Here is an example SPARQL query that uses the RDF rank value to sort results by their popularity:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
PREFIX opencyc-en: <http://sw.opencyc.org/2008/06/10/concept/en/>
SELECT * WHERE {
  ?Person a opencyc-en:Entertainer .
  ?Person rank:hasRDFRank ?rank .
}
ORDER BY DESC(?rank) LIMIT 100
```

As the sample query shows, RDF Rank weights are made available via a special system predicate. GraphDB handles triple patterns with the predicate `http://www.ontotext.com/owlim/RDFRank#hasRDFRank` in a special way, where the object of the statement pattern is bound to a literal containing the RDF Rank of the subject.

`rank#hasRDFRank` returns the rank with precision of 0.00001. `rank#hasRDFRank3`, `rank#hasRDFRank4`, and `rank#hasRDFRank5` can also be used, allowing you to retrieve the rank with precision of 0.001, 0.0001 and 0.00001 respectively. (Note that `rank#hasRDFRank5` is the same as `rank#hasRDFRank`, but has been preserved as a legacy option from previous versions of GraphDB.)

Before using this mechanism, the RDF ranks for all of a repository's nodes must be computed in advance. This is done by committing a series of SPARQL updates that use a special vocabulary to parameterize the weighting algorithm, followed by an update that triggers the computation itself.

Parameters

RDF Rank is fully controllable from *Setup* ▸ *RDF Rank*.

Parameter	Maximum iterations
Predicate	`http://www.ontotext.com/owlim/RDFRank#maxIterations`
Description	Sets the maximum number of iterations of the algorithm over all entities in the repository.
Default	20
Example	`PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>` `INSERT DATA { rank:maxIterations rank:setParam "16" . }`

Parameter	Epsilon
Predicate	`http://www.ontotext.com/owlim/RDFRank#epsilon`
Description	Terminates the weighting algorithm early when the total change of all RDF Rank scores has fallen below this value.
Default	0.00001
Example	`PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>` `INSERT DATA { rank:epsilon rank:setParam "0.00025" . }`

Full computation

To trigger the computation of the RDF Rank values for all resources, use the following update request:

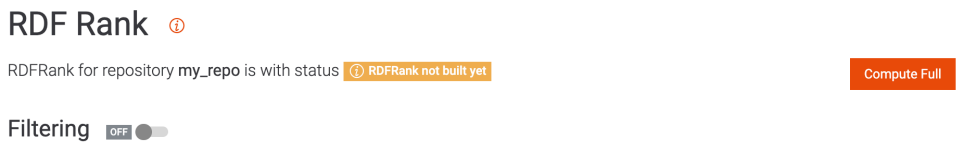
```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { _:b1 rank:compute _:b2. }
```

You can also compute the RDF Rank values in the background. This operation is asynchronous, so the plugin manager will not be blocked during execution and you can work with other plugins as the RDF Rank is being computed.

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { _:b1 rank:computeAsync _:b2. }
```

Warning: Using a SPARQL query to perform an asynchronous computation while in cluster will set your cluster out of sync. RDF Rank computations in a cluster should be performed synchronously.

Or, in Workbench, go to *Setup* ▸ *RDF Rank* and click *Compute Full*.



Note: When using the Workbench button on a standalone repository (not in a cluster), the RDF rank is computed *asynchronously*. When the button is used on a master repository (in a cluster), the rank is computed *synchronously*.

Incremental updates

The full computation of RDF Rank values for all resources can be relatively expensive. When new resources have been added to the repository after a previous full computation of the RDF Rank values, you can either have a full re-computation for all resources (see above) or compute only the RDF Rank values for the new resources (an incremental update).

The following control update computes RDF Rank values for the resources that do not have an associated value, such as the ones that have been added to the repository since the last full RDF Rank computation:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA {_:b1 rank:computeIncremental "true"}
```

Just like full computations, incremental updates can also be performed asynchronously:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA {_:b1 rank:computeIncrementalAsync "true"}
```

Warning: Using a SPARQL query to perform an asynchronous computation while in a cluster will set your cluster out of sync. RDF Rank computations in a cluster should be performed synchronously.

Note: The incremental computation uses a different algorithm, which is lightweight (in order to be fast), but is not as accurate as the proper ranking algorithm. As a result, ranks assigned by the proper and the lightweight algorithms will be slightly different.

Exporting RDF Rank values

The computed weights can be exported to an external file using an update of this form:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { _:b1 rank:export "/home/user1/rdf_ranks.txt" . }
```

If the export fails, the update throws an exception and an error message is recorded in the log file.

Checking the RDF Rank status

The RDF Rank plugin can be in one of the following statuses:

Keyword	Description
CANCELED	Rank computation was canceled
COMPUTED	Ranks are computed and up-to-date
COMPUTING	Rank computation task is currently in progress
EMPTY	There are no computed ranks
ERROR	Exception during rank computation
OUTDATED	Ranks are outdated and need re-computing
CONFIG_CHANGED	Indexing is enabled and its configuration changed since the last full computation

You can get the current status of the plugin by running the following query:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?o WHERE { ?s rank:status ?o }
```

Rank filtering

By default, the RDF Rank is calculated over the whole repository. This is useful when you want to find the most interconnected and important entities in general.

However, there are times when you are interested only in entities in certain graphs or entities related to a particular predicate. This is why the RDF Rank has a filtered mode — to filter the statements in the repository which are taken under account when calculating the rank.

You can enable the filtered mode with the following query:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { rank:filtering rank:setParam true }
```

The filtering of the statements can be performed based on predicate, graph, or type — explicit or implicit (inferred). You can make both inclusion and exclusion rules.

In order to include only statements that have a particular predicate or are in a particular named graph, add the predicate or graph IRI to one of the `includedPredicates` or `includedGraphs` lists. Empty lists are treated as wildcards. The following demonstrate how to control lists with SPARQL queries.

Get the content of a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?s WHERE { ?s rank:includedPredicates ?o }
```

Add an IRI to a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { <http:predicate> rank:includedPredicates "add" }
```

Remove an IRI from a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { <http:predicate> rank:includedPredicates "remove" }
```

The filtering can be done not only by including statements of interest but by removing them as well. In order to do so, there are two additional lists: `excludedPredicates` and `excludedGraphs`. These lists take precedence over their inclusion alternatives, so if for instance you have the same predicate in both inclusion and exclusion lists, it will be treated as excluded. These lists can be controlled in exactly the same way as the inclusion ones.

There is a convenient way to include or exclude all explicit or implicit statements. This is done with two parameters — `includeExplicit` and `includeImplicit`, which are set to `true` by default. When set to `true`, they are just disregarded and take no part in the filtering. However, if you set them to `false`, they start acting as exclusion rules and can take precedence over the inclusion lists.

You can get the status of these parameters using:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:includeExplicit _:b2 . }
```

You can set value of the parameters with:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { rank:includeExplicit rank:setParam true }
```

3.4.1.2 Prominence

The prominence for a resource is defined as the sum of the number of outgoing connections (where the resource is the subject of a triple) and the number of incoming connections (where the resource is the object of a triple). GraphDB maintains the numbers automatically.

Examples

- Retrieve the prominence for a given resource:

```
SELECT ?prominence {
  <http://example.com/Book1> <http://www.ontotext.com/owlim/entity#hasProminence> ?
↪prominence
}
```

- Filter bound resources by prominence:

```
SELECT ?book {
  ?book a <http://example.com/Book> ;
        <http://www.ontotext.com/owlim/entity#hasProminence> 5
}
```

- Filter all resources by prominence:

```
SELECT ?node {
  ?node <http://www.ontotext.com/owlim/entity#hasProminence> 10
}
```

- Retrieve all resources and their prominence:

```
SELECT ?node ?prominence {
  ?node <http://www.ontotext.com/owlim/entity#hasProminence> ?prominence
}
```

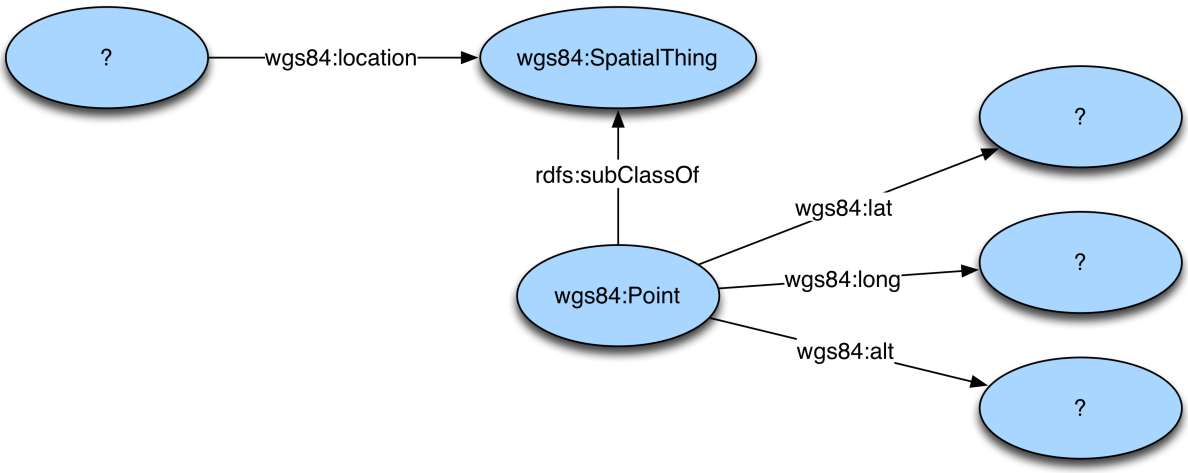
This feature is implemented by the Expose Entity plugin.

3.4.2 Geospatial extensions

3.4.2.1 What are geospatial extensions

GraphDB provides support for two-dimensional geospatial data that uses the WGS84 Geo Positioning RDF vocabulary (World Geodetic System 1984). Specialized indexes can be used for this type of data, which allow efficient evaluation of query forms and extension functions for finding locations:

- within a certain distance of a point — that is, within a specified circle on the surface of a sphere (Earth), using the nearby(…) construction
- within rectangles and polygons, where the vertices are defined by spherical polar coordinates, using the within(…) construction



The WGS84 ontology contains several classes and predicates:

Element	Description
`SpatialThing`	A class for representing anything with a spatial extent: size, shape, or position.
`Point`	A class for representing a point (relative to Earth) defined by latitude-longitude and altitude. `subClassOf http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing`
`location`	The relation between a thing and where it is. Range `SpatialThing subPropertyOf http://xmlns.com/foaf/0.1/based_near`
`lat`	The WGS84 latitude of a SpatialThing in decimal degrees. `domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing`
`long`	The WGS84 longitude of a SpatialThing in decimal degrees. `domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing`
`lat_long`	A comma-separated representation of a latitude-longitude coordinate.
`alt`	The WGS84 altitude of a SpatialThing as decimal meters above the local reference ellipsoid. `domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing`

3.4.2.2 Creating a geospatial index

Execute the following `INSERT` query:

```
PREFIX ontogeo: <http://www.ontotext.com/owlim/geo#>
INSERT DATA { _:b1 ontogeo:createIndex _:b2. }
```

If all geospatial data is indexed successfully, the above update query will succeed. If there is an error, you will get a notification about a failed transaction and an error will be registered in the GraphDB log files.

Note: If there is no geospatial data in the repository (no statements describing resources with latitude and longitude properties), this update query will fail.

3.4.2.3 Geospatial query syntax

The Geospatial query syntax is the SPARQL RDF Collections syntax. It uses round brackets as a shorthand for the statements, which connect a list of values using `rdf:first` and `rdf:rest` predicates with terminating `rdf:nil`. Statement patterns that use custom geospatial predicates supported by GraphDB are treated differently by the query engine.

The following special syntax is supported when evaluating SPARQL queries. All descriptions use the namespace: `omgeo: <http://www.ontotext.com/owlim/geo#>`

Construct	Nearby (lat long distance)
Syntax	`?point omgeo:nearby(?lat ?long ?distance)`
Description	This statement pattern will evaluate to `true` if the following constraints hold: • `?point geo:lat ?plat .` • `?point geo:long ?plong .` • Shortest great circle distance from `(?plat, ?plong) to (?lat, ?long) <= ?distance` Such a construction uses the geospatial indexes to find bindings for `?point`, which lie within the defined circle. Constants are allowed for any of `?lat ?long ?distance`, where latitude and longitude are specified in decimal degrees and distance is specified in either kilometers ('km' suffix) or miles ('mi' suffix). If the units are not specified, then 'km' is assumed.
Restrictions	Latitude is limited to the range -90 (South) to 90 (North). Longitude is limited to the range -180 (West) to +180 (East).

Examples	Find the names of airports within 50 miles of Seoul:
	`PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>`
	`PREFIX geo-ont: <http://www.geonames.org/ontology#>`
	`PREFIX omgeo: <http://www.ontotext.com/owlim/geo#>`
	
	`SELECT distinct ?airport`
	`WHERE {`
	`?base geo-ont:name "Seoul" .`
	`?base geo-pos:lat ?latBase .`
	`?base geo-pos:long ?longBase .`
	`?link omgeo:nearby(?latBase ?longBase "50mi") .`
	`?link geo-ont:name ?airport .`
	`?link geo-ont:featureCode geo-ont:S.AIRP .`
	`}`

Construct	Within (rectangle)
Syntax	`?point omgeo:within(?lat1 ?long1 ?lat2 ?long2)`
Description	This statement pattern is used to test or find points that lie within the rectangle specified by diagonally opposite corners `?lat1 ?long1` and `?lat2 ?long2`. The corners of the rectangle must be either constants or bound values. It will evaluate to `true`, if the following constraints hold: • `?point geo:lat ?plat .` • `?point geo:long ?plong .` • `?lat1 <= ?plat <= ?lat2` • `?long1 <= ?plong <= ?long2` Note that the most westerly and southerly corners must be specified first and the most northerly and easterly ones second. Constants are allowed for any of `?lat1 ?long1 ?lat2 ?long2`, where latitude and longitude are specified in decimal degrees. If `?point` is unbound, then bindings for all points within the rectangle will be produced. Rectangles that span across the +/-180 degree meridian might produce incorrect results.
Restrictions	Latitude is limited to the range -90 (South) to +90 (North). Longitude is limited to the range -180 (West) to +180 (East). Rectangle vertices must be specified in the order lower-left followed by upper-right.

Examples	Find tunnels lying within a rectangle enclosing Tirol, Austria:
	`PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>`
	`PREFIX geo-ont: <http://www.geonames.org/ontology#>`
	`PREFIX omgeo: <http://www.ontotext.com/owlim/geo#>`
	
	`SELECT ?feature ?lat ?long`
	`WHERE {`
	`?link omgeo:within(45.85 9.15 48.61 13.18) .`
	`?link geo-ont:featureCode geo-ont:R.TNL .`
	`?link geo-ont:name ?feature .`
	`?link geo-pos:lat ?lat .`
	`?link geo-pos:long ?long .`
	`}`

Construct	Within (polygon)
Syntax	`?point omgeo:within(?lat1 ?long1 ... ?latN ?longN)`
Description	This statement pattern is used to test or find points that lie within the polygon whose vertices are specified by three or more latitude-longitude pairs. The values of the vertices must be either constants or bound values. It will evaluate to `true`, if the following constraints hold: • `?point geo:lat ?plat .` • `?point geo:long ?plong .` • the position `?plat ?plong` is enclosed by the polygon The polygon is closed automatically if the first and last vertices do not coincide. The vertices must be constants or bound values. Coordinates are specified in decimal degrees. If `?point` is unbound, then bindings for all points within the polygon will be produced.
Restrictions	Latitude is limited to the range -90 (South) to +90 (North). Longitude is limited to the range -180 (West) to +180 (East).

Examples	Find caves in the sides of cliffs lying within a polygon approximating the shape of England:
	`PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>`
	`PREFIX geo-ont: <http://www.geonames.org/ontology#>`
	`PREFIX omgeo: <http://www.ontotext.com/owlim/geo#>`
	`SELECT ?feature ?lat ?long`
	`WHERE {`
	`?link omgeo:within( "51.45" "-2.59"`
	`"54.99" "-3.06"`
	`"55.81" "-2.03"`
	`"52.74" "1.68"`
	`"51.17" "1.41" ) .`
	`?link geo-ont:featureCode geo-ont:S.CAVE .`
	`?link geo-ont:name ?feature .`
	`?link geo-pos:lat ?lat .`
	`?link geo-pos:long ?long .`
	`}`

3.4.2.4 Extension query functions

At present, there is just one SPARQL extension function. The prefix `omgeo:` stands for the namespace `<http://www.ontotext.com/owlim/geo#>`.

Function	Distance function
Syntax	`xsd:double omgeo:distance(numeric lat1, numeric long1, numeric lat2, numeric long2)`
Description	This SPARQL extension function computes the distance between two points in kilometers and can be used in `FILTER` and `ORDER BY` clauses.
Restrictions	Latitude is limited to the range -90 (South) to +90 (North). Longitude is limited to the range -180 (West) to +180 (East).
Examples	Find airports within 80 miles of Bournemouth airport. These airports have to meet the specified filter criteria — the distance from them to the Brize Norton airport is under 80 kilometers (note the combination of a miles value and kilometers value in the criteria) and are ordered by the distance in ascending order.

```
PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX geo-ont: <http://www.geonames.org/ontology#>
PREFIX omgeo:   <http://www.ontotext.com/owlim/geo#>

SELECT distinct ?airport_name
WHERE {
  ?a1 geo-ont:name "Bournemouth" .
  ?a1 geo-pos:lat ?lat1 .
  ?a1 geo-pos:long ?long1 .
  ?airport omgeo:nearby(?lat1 ?long1 "80mi" ) .
  ?airport geo-ont:name ?airport_name .
  ?airport geo-ont:featureCode geo-ont:S.AIRP .
  ?airport geo-pos:lat ?lat2 .
  ?airport geo-pos:long ?long2 .
  ?a2 geo-ont:name "Brize Norton" .
  ?a2 geo-pos:lat ?lat3 .
  ?a2 geo-pos:long ?long3 .
  FILTER( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) < 80)
}
ORDER BY ASC( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) )
```

3.4.2.5 Implementation details

Knowing the implementation's algorithms and assumptions allow you to make the best use of the GraphDB geospatial extensions.

The following aspects are significant and can affect the expected behavior during query answering:

- Spherical Earth — the current implementation treats the Earth as a perfect sphere with a 6371.009km radius.
- Only two-dimensional points are supported; there is no special handling of `geo:alt` (meters above the reference surface of the Earth).
- All latitude and longitude values must be specified using decimal degrees, where East and North are positive and -90 <= latitude <= +90 and -180 <= longitude <= +180.
- Distances must be in units of kilometers (suffix 'km') or statute miles (suffix 'mi'). If the suffix is omitted, kilometers are assumed.
- `omgeo:within( rectangle )` construct uses a 'rectangle' whose edges are lines of latitude and longitude, so the north-south distance is constant, and the rectangle described forms a band around the Earth, which starts and stops at the given longitudes.
- `omgeo:within( polygon )` joins vertices with straight lines on a cylindrical projection of the Earth tangential to the equator. A straight line starting at the point under test and continuing East out of the polygon is examined to see how many polygon edges it intersects. If the number of intersections is even, then the point is outside the polygon. If the number of intersections is odd, the point is inside the polygon. With the current algorithm, the order of vertices is not relevant (clockwise or anticlockwise).
- `omgeo:within()` may not work correctly when the region (polygon or rectangle) spans the +/-180 meridian.

- `omgeo:nearby()` uses the great circle distance between points.

3.4.3 GeoSPARQL support

3.4.3.1 What is GeoSPARQL

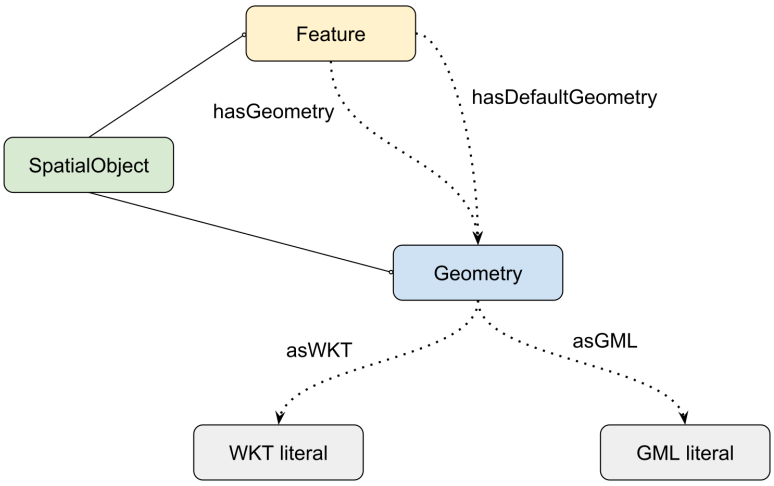
GeoSPARQL is a standard for representing and querying geospatial linked data from the Open Geospatial Consortium (OGC). The standard provides:

- A small topological ontology in RDFS/OWL for representation using Geography Markup Language (GML) and Well-Known Text (WKT) literals.
- Simple Features, RCC8, and Egenhofer topological relationship vocabularies and ontologies for qualitative reasoning.
- A SPARQL query interface using a set of topological SPARQL extension functions for quantitative reasoning.

The GraphDB GeoSPARQL plugin enables the conversion of Well-Known Text from different coordinate reference systems (CRS) into the CRS84 format, which is the default CRS according to the Open Geospatial Consortium (OGC). The plugin will properly index data of all known CRS types and can be queried in the default CRS84 format.

Note: The GraphDB GeoSPARQL plugin implements the GeoSPARQL 1.0 Standard specification. For more information on GeoSPARQL predicates and functions, see the official specification.

The following is a simplified diagram of the GeoSPARQL classes Feature and Geometry, as well as some of their properties:



3.4.3.2 Usage

Configuration parameters

The following parameters can be used when configuring the plugin:

Parameter	enabled
Predicate	`<http://www.ontotext.com/plugins/geosparql#enabled>`
Description	Enables and disables plugin.
Default	`false`
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:enabled "true" . }`

Parameter	prefixTree
Predicate	`<http://www.ontotext.com/plugins/geosparql#prefixTree>`
Description	Implementation of the tree used while building the index; stores value before rebuilding.
Default	prefixTree.QUAD
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:prefixTree "geohash" . }`

Parameter	precision
Predicate	`<http://www.ontotext.com/plugins/geosparql#precision>`
Description	Specifies the desired precision; stores value before rebuilding.
Default	11 min value 1; maximum value depends on used prefixTree or (24 for geohash and 50 for QUAD)
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:precision "11" . }`

Parameter	currentPrefixTree
Predicate	`<http://www.ontotext.com/plugins/geosparql#currentPrefixTree>`
Description	Value of last built index.
Default	PrefixTree.QUAD
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:currentPrefixTree "geohash" . }`

Parameter	currentPrecision
Predicate	`<http://www.ontotext.com/plugins/geosparql#currentPrecision>`
Description	Value of last built index.
Default	11
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:currentPrecision "11" . }`

Parameter	maxBufferedDocs
Predicate	`<http://www.ontotext.com/plugins/geosparql#maxBufferedDocs>`
Description	Speeds up building and rebuilding of index.
Default	1,000 (maximum allowed 5,000)
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:maxBufferedDocs "3000" . }`

Parameter	ramBufferSizeMB
Predicate	`<http://www.ontotext.com/plugins/geosparql#ramBufferSizeMB>`
Description	Speeds up building and rebuilding of index.
Default	32.0 (maximum allowed 512.0)
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:ramBufferSizeMB "256.0" . }`

Parameter	ignoreErrors
Predicate	`<http://www.ontotext.com/plugins/geosparql#ignoreErrors>`
Description	Ensures building of the index even in case of erroneous data.
Default	`false`
Example	`PREFIX geoSparql: <http://www.ontotext.com/plugins/geosparql#>` `INSERT DATA { [] geoSparql:ignoreErrors "true" . }`

Plugin control predicates

The plugin allows you to configure it through SPARQL `UPDATE` queries with embedded control predicates.

Enable plugin

When the plugin is enabled, it indexes all existing GeoSPARQL data in the repository and automatically re-indexes any updates.

```
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
  [] geosparql:enabled "true" .
}
```

Note: All functions require WKT or GML literals as input while the predicates expect resources of type `geo:Feature` or `geo:Geometry`. The GraphDB implementation has a non-standard extension that also lets you use literals with the predicates. See *Example 2 (using predicates)* for an example of that usage.

Warning: All GeoSPARQL functions that use the `geof:` prefix (for example, `geof:sfOverlaps`) do not use any indexes and are always enabled. This is why it's best to use the indexed operations such as `geo:sfOverlaps` whenever possible.

Disable plugin

When the plugin is disabled, it does not index any data or process updates. It also does not handle any of the GeoSPARQL predicates.

```
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
  [] geosparql:enabled "false" .
}
```

Check the current configuration

All the plugin configuration parameters are stored in `$GDB_HOME/data/repositories/<repoId>/storage/GeoSPARQL/config.properties`. You can use the following query to check the current runtime configuration:

```
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

SELECT DISTINCT * WHERE {
    [] geosparql:currentPrefixTree ?tree;
        geosparql:currentPrecision ?precision;
}
```

Update the current configuration

The plugin supports two indexing algorithms: quad prefix tree and geohash prefix tree. Both algorithms support approximate matching controlled with the precision parameter. The default precision value of `11` for the quad prefix is about ±2.5km on the equator. When increased to `20` the precision goes down to ±6m accuracy. Respectively, the geohash prefix tree with precision `11` results ±1m.

```
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
    [] geosparql:prefixTree "quad"; #geohash
        geosparql:precision "25".
}
```

After changing the indexing algorithm a reindex is required.

Speed up the building and rebuilding of the GeoSPARQL index

To speed up the building and rebuilding of your GeoSPARQL index, we recommend setting higher values for the `ramBufferSizeMB` and `maxBufferedDocs` parameters. This disables the Lucene IndexWriter autocommit and starts flushing disk changes if one of these values is reached.

Default and maximum values are as follows:

- `ramBufferSizeMB` — default 32.0, maximum 512.0.
- `maxBufferedDocs` — default 1,000, maximum 5,000.

Depending on your dataset and machine parameters, you can experiment with the values to find the ones most suitable for your use case.

Note: Do not set these values too high or you may hit an IndexWriter over-merging issue.

Force reindex geometry data

This configuration option is usually used after a configuration change or when index files are either corrupted or have been mistakenly deleted.

```
PREFIX onto-geo: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
    [] onto-geo:forceReindex []
}
```

Ignore errors on indexing

```
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
    [] geosparql:ignoreErrors "true"
}
```

The `ignoreErrors` predicate determines whether the GeoSPARQL index will continue building if an error has occurred. If the value is set to `false`, the whole index will fail if there is a problem with a document. If the value is set to `true`, the index will continue building and a warning will be logged. By default, the value of `ignoreErrors` is `false`.

GeoSPARQL extensions

On top of the standard GeoSPARQL functions, GraphDB adds a few useful extensions based on the USeekM library. The prefix `geoext:` stands for the namespace `<http://rdf.useekm.com/ext#>`.

Types such as `geo:Geometry` and `geo:Point` refer to GeoSPARQL types in the `http://www.opengis.net/ont/geosparql#` namespace.

Function	Description
`xsd:double geoext:area(geomLiteral g)`	Calculates the area of the surface of the geometry.
`geomLiteral geoext:closestPoint(geomLiteral g1, geomLiteral g2)`	For two given geometries, computes the point on the first geometry that is closest to the second geometry.
`xsd:boolean geoext:containsProperly(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry properly contains the second geometry. *Geom1* contains properly *geom2* if all *geom1* contains *geom2* and the boundaries of the two geometries *do not* intersect.
`xsd:boolean geoext:coveredBy(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry is covered by the second geometry. *Geom1* is covered by *geom2* if every point of *geom1* is a point of *geom2*.
`xsd:boolean geoext:covers(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry covers the second geometry. *Geom1* covers *geom2* if every point of *geom2* is a point of *geom1*.
`xsd:double geoext:hausdorffDistance(geomLiteral g1, geomLiteral g2)`	Measures the degree of similarity between two geometries. The measure is normalized to lie in the range [0, 1]. Higher measures indicate a greater degree of similarity.
`geo:Line geoext:shortestLine(geomLiteral g1, geomLiteral g2)`	Computes the shortest line between two geometries and returns it as a LineString object.
`geomLiteral geoext:simplify(geomLiteral g, double d)`	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm.
`geomLiteral geoext:simplifyPreserveTopology(geomLiteral g, double d)`	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm. This will avoid the creation of derived geometries (polygons in particular) that are invalid.
`xsd:boolean geoext:isValid(geomLiteral g)`	Checks whether the input geometry is a valid geometry.

3.4.3.3 GeoSPARQL examples

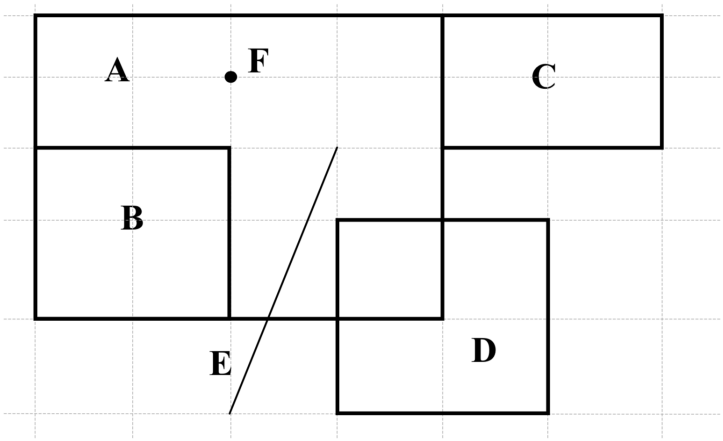
This section contains examples of SELECT queries on geographic data.

Examples 1, 2, and 3 have a variant using a function (corresponding to the same example in the GeoSPARQL specification), as well as a variant where the function is substituted with a predicate. Examples 4 and 5 use a predicate and correspond to the same examples in the specification.

To run the examples, you need to:

- Download and import the file `geosparql-example.rdf`.
- Enable the GeoSPARQL plugin.

The data defines the following spatial objects:



Example 1

Find all features that feature `my:A` contains, where spatial calculations are based on `my:hasExactGeometry`.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    my:A my:hasExactGeometry ?aGeom .
    ?aGeom geo:asWKT ?aWKT .
    ?f my:hasExactGeometry ?fGeom .
    ?fGeom geo:asWKT ?fWKT .
    FILTER (geof:sfContains(?aWKT, ?fWKT) && !sameTerm(?aGeom, ?fGeom))
}
```

Using a predicate

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    my:A my:hasExactGeometry ?aGeom .
    ?f my:hasExactGeometry ?fGeom .
    ?aGeom geo:sfContains ?fGeom .
    FILTER (!sameTerm(?aGeom, ?fGeom))
}
```

Example 1 result

?f
my:B
my:F

Example 2

Find all features that are within a transient bounding box geometry, where spatial calculations are based on `my:hasPointGeometry`.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    ?f my:hasPointGeometry ?fGeom .
    ?fGeom geo:asWKT ?fWKT .
    FILTER (geof:sfWithin(?fWKT, '''
        <http://www.opengis.net/def/crs/OGC/1.3/CRS84>
            Polygon ((-83.4 34.0, -83.1 34.0,
                      -83.1 34.2, -83.4 34.2,
                      -83.4 34.0))
        '''^^geo:wktLiteral))
}
```

Using a predicate

Note: Using geometry literals in the object position is a GraphDB extension and not part of the GeoSPARQL specification.

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
   ?f my:hasPointGeometry ?fGeom .
   ?fGeom geo:sfWithin '''
        <http://www.opengis.net/def/crs/OGC/1.3/CRS84>
            Polygon ((-83.4 34.0, -83.1 34.0,
                      -83.1 34.2, -83.4 34.2,
                      -83.4 34.0))
        '''^^geo:wktLiteral
}
```

Example 2 result

?f
my:D

Example 3

Find all features that touch the union of feature `my:A` and feature `my:D`, where computations are based on `my:hasExactGeometry`.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    ?f my:hasExactGeometry ?fGeom .
    ?fGeom geo:asWKT ?fWKT .
    my:A my:hasExactGeometry ?aGeom .
    ?aGeom geo:asWKT ?aWKT .
    my:D my:hasExactGeometry ?dGeom .
    ?dGeom geo:asWKT ?dWKT .
    FILTER (geof:sfTouches(?fWKT, geof:union(?aWKT, ?dWKT)))
}
```

Using a predicate

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    ?f my:hasExactGeometry ?fGeom .
    ?fGeom geo:asWKT ?fWKT .
    my:A my:hasExactGeometry ?aGeom .
    ?aGeom geo:asWKT ?aWKT .
    my:D my:hasExactGeometry ?dGeom .
    ?dGeom geo:asWKT ?dWKT .
    BIND(geof:union(?aWKT, ?dWKT) AS ?union) .
    ?fGeom geo:sfTouches ?union
}
```

Example 3 result

?f
my:C

Example 4

Find the 3 closest features to feature `my:C`, where computations are based on `my:hasExactGeometry`.

```
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
    my:C my:hasExactGeometry ?cGeom .
    ?cGeom geo:asWKT ?cWKT .
    ?f my:hasExactGeometry ?fGeom .
    ?fGeom geo:asWKT ?fWKT .
    FILTER (?fGeom != ?cGeom)
}
ORDER BY ASC(geof:distance(?cWKT, ?fWKT, uom:metre))
LIMIT 3
```

Example 4 result

?f
my:A
my:E
my:D

Note: The example in the GeoSPARQL specification has a different order in the result: `my:A`, `my:D`, `my:E`. In fact, feature `my:E` is closer than feature `my:D` even if that does not seem obvious from the drawing of the objects. `my:E`'s closest point is 0.1° to the West of `my:C`, while `my:D`'s closest point is 0.1° to the South. At that latitude and longitude the difference in terms of distance is larger in latitude, hence `my:E` is closer.

Example 5

Find all features or geometries that overlap feature `my:A`.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX my: <http://example.org/ApplicationSchema#>

SELECT ?f
WHERE {
    ?f geo:sfOverlaps my:AExactGeom
}
```

Example 5 result

?f
my:D
my:DExactGeom

Note: The example in the GeoSPARQL specification has additional results `my:E` and `my:EExactGeom`. In fact, `my:E` and `my:EExactGeom` do not overlap `my:AExactGeom` because they are of different dimensions (`my:AExactGeom` is a Polygon and `my:EExactGeom` is a LineString) and the *overlaps* relation is defined only for objects of the same dimension.

3.4.4 Full-text search

Full-text search (FTS) indexing enables very fast queries over textual data. Typically, FTS is used to retrieve data that represents text written in a human language such as English, Spanish, or French.

GraphDB supports various mechanisms for performing full-text search depending on the use case and the needs of a given project.

3.4.4.1 Full-text search using the GraphDB connectors

GraphDB connectors can index, search, and retrieve entire documents composed of a set of RDF statements. To do this, those connectors need a predefined data model that describes how every indexed document is constructed from a template of RDF statements. Queries can search in one or more document fields defined by the data model, and results return document IDs to show which documents were found.

See more about the *full-text search with the GraphDB connectors*, as well as the *Lucene*, *Solr*, *Elasticsearch*, and *OpenSearch* connectors.

3.4.4.2 Simple full-text search index

GraphDB 10.1 introduced a simple FTS index that covers some basic FTS use cases. This index contains literals and IRIs:

- There is no data model, so it is easy to set up.
- Queries search in literals and IRIs.
- Results return the matching literals and IRIs.

How the search works

In general, searching is performed via SPARQL using a pattern like this:

```
?value onto:fts (query index limit)
```

There are three search arguments:

- The **query**: string or language-tagged string, required
- The **index to search**: string, optional
- The **limit of the search**: integer, optional

The matching values will be returned as bindings of the provided variable in the triple pattern's subject position (`?value` in the model above).

When no index is supplied as a parameter, GraphDB FTS will use the following:

- If the query is a plain string without a language tag, then the index will be the configured index for string literals (via the *Enable full-text search (FTS) index* repository configuration parameter).
- If the query is a language-tagged string, then the language tag will be used to determine the index name.

Note: When an index is supplied as a parameter, the language tag of the query string will be ignored.

When only the query is provided (the only required argument), it's best to provide it directly without constructing an RDF list. So, the pattern can be simplified to:

```
?value onto:fts query
```

Some examples:

- `("query" "en" 10)`: Search for "query" in the "en" index and limit results to 10.
- `("query" 15)`: Search for "query" in the index configured via fts-string-literals-index and limit results to 15.
- `("query"@de 20)`: Search for "query" in the "de" index and limit results to 20.
- `("query"@de-CH 20)`: Search for "query" in the "de" index and limit results to 20. Note that only the language part of the tag de-CH determines the index.
- `("query" "fr")`: Search for "query" in the "fr" index and do not apply a limit.
- `"query"@fr`: Search for "query" in the "fr" index and do not apply a limit — when a sole argument is provided, it does not need to be inside an RDF list.

Query syntax

The queries are parsed using Lucene's StandardQueryParser class.

A query consists of clauses, field specifications, grouping and boolean operators, and interval functions.

Note: Keep in mind these details in particular:

- **Field specifications**: There are no other field names but the default field name so there is no valid case where the user must specify a field name.
- **Escaping in SPARQL**: All query syntax examples specify the expected Lucene query string. If you provide these strings as SPARQL literals, you may need to escape `"` and `\` as required by SPARQL.

Warning: FTS performs no text analysis on some of the specialized query types. Lexical analysis is only run on complete terms (a term or phrase query). Query types containing incomplete terms such as prefix, wildcard, regex, or fuzzy queries skip the analysis stage and get directly added to the query tree. The only transformation applied to partial query terms is lowercasing.

This may lead to surprising results if you expect stemming or lemmatization. For example, searching for "resti*" and expecting to find "resting" will not work when using the English analyzer since the word "resting" was analyzed and indexed as "rest".

Basic clauses

A query must contain one or more clauses. A clause can be a literal term, a phrase, a wildcard expression, or any supported expression.

The following are some examples of simple one-clause queries:

Query	Description
`test`	Selects documents containing the word “test” (term clause).
`"test equipment"`	Phrase search; selects documents containing the phrase “test equipment” (phrase clause).
`"test failure"~4`	Proximity search; selects documents containing the words “test” and “failure” within 4 words (positions) from each other. The provided “proximity” is technically translated into the “edit distance” (the maximum number of atomic word-moving operations required to transform the document’s phrase into the query phrase).
`tes*`	Prefix wildcard matching; selects documents containing words starting with “tes”, such as: “test”, “testing” or “testable”.
`/(p|n).st/`	Documents containing word roots matching the provided regular expression, such as “post” or “nest”.
`nest~2`	Fuzzy term matching; documents containing words within 2-edits distance (2 additions, removals, or replacements of a letter) from “nest”, such as “test”, “net”, or “rests”.

Boolean operators and grouping

You can combine clauses using boolean AND, OR, and NOT operators to form more complex expressions, for example:

Query	Description
`test AND results`	Selects documents containing both the word “test” and the word “results”.
`test OR suite OR results`	Selects documents with at least one of “test”, “suite”, or “results”.
`test AND NOT complete`	Selects documents containing “test” and not containing “complete”.
`test AND (pass* OR fail*)`	Grouping; use parentheses to specify the precedence of terms in a boolean clause. Query will match documents containing “test” and a word starting with “pass” or “fail”.
`(pass fail skip)`	Shorthand notation; documents containing at least one of “pass”, “fail”, or “skip”.

Note: The boolean operators must be written in all caps, otherwise they are parsed as regular terms.

Range operators

To search for ranges of textual or numeric values, use square or curly brackets, for example:

Query	Description
`[Jones TO Smith]`	Inclusive range; selects documents that contain any value between “Jones” and “Smith”, including boundaries.
`{Jones TO Smith}`	Exclusive range; selects documents that contain any value between “Jones” and “Smith”, excluding boundaries.
`{Jones TO *]`	One-sided range; selects documents that contain any value larger than (that is, sorted after) “Jones”.

Note: These will work intuitively only with the "iri" index, e.g., "`[http://www.w3.org/2000/01/rdf-schema#comment TO http://www.w3.org/2000/01/rdf-schema#range]`" will retrieve all IRIs that are alphabetically ordered between `http://www.w3.org/2000/01/rdf-schema#comment` and `http://www.w3.org/2000/01/rdf-schema#range` inclusive. If used with any of the other indexes, they will return matches but it will not be intuitive what they match.

Term boosting

Terms, quoted terms, term range expressions, and grouped clauses can have a floating-point weight boost applied to them to increase their score relative to other clauses. For example:

Query	Description
`jones^2 OR smith^0.5`	Prioritize documents with "jones" term over matches on the "smith" term.
`(a OR b NOT c)^2.5 OR d`	Apply the boost to a sub-query.

Special character escaping

Most search terms can be put in double quotes, making special character escaping not necessary. If the search term contains the quote character (or cannot be quoted for some reason), any character can be quoted with a backslash. For example:

Query	Description
`\:\(quoted\+term\)\:`	A single search term `(quoted+term):` with escape sequences. An alternative quoted form would be simpler: `":(quoted+term):"`.

Minimum-should-match constraint for boolean disjunction groups

A minimum-should-match operator can be applied to a disjunction boolean query (a query with only "OR"-subclauses) to force the query to match documents with at least the provided number of these subclauses. For example:

Query	Description
`(blue crab fish)@2`	Matches all documents with at least two terms from the set [blue, crab, fish] in any order.
`((yellow OR blue) crab fish)@2`	Sub-clauses of a boolean query can themselves be complex queries; here the min-should-match selects documents that match at least two of the provided three sub-clauses.

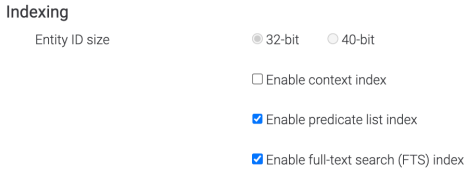
Interval function clauses

Interval functions are a powerful tool for expressing search needs in terms of one or more contiguous fragments of text and their relationship to one another. All interval clauses start with the `fn:` prefix. For example:

Query	Description
`fn:ordered(quick brown fox)`	Matches all documents with at least one ordered sequence of "quick", "brown", and "fox" terms.
`fn:maxwidth(5 fn:atLeast(2 quick brown fox))`	Matches all documents where at least two of the three terms "quick", "brown", and "fox" occur within five positions of each other.

Common use cases

Before performing a full-text search you must enable the FTS index. This can be done at *repository creation* by setting the *Enable full-text search (FTS) index* to `true`, as well as at a later stage if you want to edit the repository configuration.



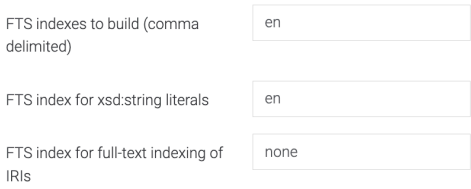
Single language

Let's say that your data is in a single supported language and you want to perform full-text search in order to find literals that match. Literals may or may not have a language tag; for example:

- "This is a literal in English without a language tag"
- "This is another literal in English with a language tag for the language only"**@en**
- "This is yet another literal tagged for English in Canada"**@en-CA**

To configure the sample search:

1. Create a repository.
2. In its configuration menu, enable the `"en"` index by setting *FTS indexes to build* to "en".
3. Literals with no language tag must also go into the `"en"` index, so we set *FTS index for xsd:string literals* to "en".



Important: After each change applied to any FTS parameters you must *restart the repository*.

To try this configuration out, insert some tagged sample data by executing the following update request in the Workbench SPARQL editor:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <urn:d1> rdfs:label "This is a literal in English without a language tag",
            "This is another literal in English with a language tag for the language only"@en,
            "This is yet another literal tagged for English in Canada"@en-CA,
            "Let's pretend this literal isn't in English by tagging it as German"@de
}
```

So if we run the following example query against it:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # Note that this exploits the fact that we haven't enabled the default index,
    # so the index for indexing string literals (en) is the default query index
```

(continues on next page)

(continued from previous page)

```
    ?value onto:fts "english literal"
}
```

Or this one:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The language tag of the query literal supplies the index to query
    ?value onto:fts "english literal"@en
}
```

Or this one:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The query string and the index to query are supplied as two separate values
    # inside an RDF list
    ?value onto:fts ("english literal" "en")
}
```

They will all return the three literals that are not tagged as German:

	value
1	"This is a literal in English without a language tag"
2	"This is yet another literal tagged for English in Canada"@en-CA
3	"This is another literal in English with a language tag for the language only"@en

Multiple languages

The strings in the following bulleted list are in English or German and we want to perform full-text search in order to find literals that match. Literals with no language tag are in one of the desired languages: English.

- "This is a literal in English without a language tag"
- "This is another literal in English with a language tag for the language only"**@en**
- "This is yet another literal tagged for English in Canada"**@en-CA**
- "Das ist ein schönes deutsches Literal"**@de**
- "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"**@de-CH**

To configure a search through this set of literals:

1. Create a repository.
2. In its configuration menu, enable the `"en"` and `"de"` indexes by setting *FTS indexes to build* to "en, de". You can extend this with additional languages by adding them to the list.
3. Literals with no language tag must also go into the `"en"` index, so set *FTS index for xsd:string literals* to "en".

FTS indexes to build (comma delimited)	en, de
FTS index for xsd:string literals	en
FTS index for full-text indexing of IRIs	none

The following update request inserts the data shown above into the repository:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <urn:d2> rdfs:label "This is a literal in English without a language tag",
            "This is another literal in English with a language tag for the language only"@en,
            "This is yet another literal tagged for English in Canada"@en-CA,
            "Das ist ein schönes deutsches Literal"@de,
            "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"@de-CH
}
```

Searching in English is exactly the same as in the first use case above. To search the additional German index, we specify it like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The language tag of the query literal supplies the index to query
    ?value onto:fts "deutsch literal"@de
}
```

Or this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The query string and the index to query are supplied as two separate values
    # inside an RDF list
    ?value onto:fts ("deutsch literal" "de")
}
```

Both of these queries will return the two German literals.

	value
1	"Das ist ein schönes deutsches Literal"@de
2	"Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"@de-CH

Note: Keep in mind that if you have other data in the repository, it may affect the results.

Ignore untagged literals

In this case, our data is in one or more supported languages (English and German) and we want to perform full-text search in order to find literals that match. Literals with no language tag should not be treated as any of those languages and need not be searched. Data may look like any of these:

- "This is another literal in English with a language tag for the language only"**@en**
- "This is yet another literal tagged for English in Canada"**@en-CA**
- "Das ist ein schönes deutsches Literal"**@de**
- "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"**@de-CH**
- "This is a literal in English without a language tag" **(this must not be indexed)**

To configure the sample search:

1. Create a repository.
2. In its configuration menu, enable the `"en"` and `"de"` indexes by setting *FTS indexes to build* to "en, de". You can extend this with additional languages by adding them to the list.
3. Literals with no language tag need to not be indexed, so we set *FTS index for xsd:string literals* to "none".

FTS indexes to build (comma delimited)	en, de
FTS index for xsd:string literals	none
FTS index for full-text indexing of IRIs	none

The following update request inserts the data shown above into the repository:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <urn:d3> rdfs:label "This is another literal in English with a language tag for the language only
↪"@en,
            "This is yet another literal tagged for English in Canada"@en-CA,
            "Das ist ein schönes deutsches Literal"@de,
            "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"@de-CH,
            "This is a literal in English without a language tag"
}
```

Searching in any of the languages requires the query to to specify the index. There is no default search index because the *FTS index for xsd:string literals* is set to "none":

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The language tag of the query literal supplies the index to query
    ?value onto:fts "english literal"@en
}
```

Or this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The query string and the index to query are supplied as two separate values
    # inside an RDF list
    ?value onto:fts ("english literal" "en")
}
```

Both queries will return the two literals that are tagged for English but not the untagged one.

	value
1	"This is yet another literal tagged for English in Canada"@en-CA
2	"This is another literal in English with a language tag for the language only"@en

Untagged literals not treated as any language but still searchable

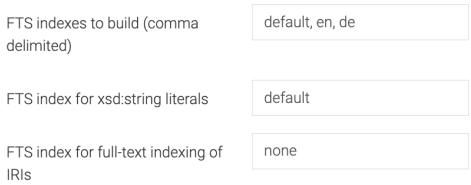
Here, our data is in one or more supported languages (English and German) and we want to perform full-text search in order to find literals that match.

Literals with no language tag should not be treated as any of those languages but should provide language-agnostic full-text search. These literals may be data like UUIDs or anything else that has a textual representation that we may want to search. Data may look like any of these:

- "This is another literal in English with a language tag for the language only"**@en**
- "This is yet another literal tagged for English in Canada"**@en-CA**
- "Das ist ein schönes deutsches Literal"**@de**
- "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"**@de-CH**
- "96ac1c60-7997-45a3-8dfe-b57b24c1cb62" **(this will be indexed separately)**

To configure the sample search:

1. Create a repository.
2. In its configuration menu, enable the `"default"` index, as well as the indexes `"en"` and `"de"`, by setting *FTS indexes to build* to "default, en, de". You can extend this with additional languages by adding them to the list.
3. Literals with no language tag must be indexed in a language-agnostic manner, so we set *FTS index for xsd:string literals* to "default" (which is also the default value for that repository configuration property).



Important: The values of *FTS indexes to build* must contain the values for *FTS index for xsd:string literals* and *FTS index for full-text indexing of IRIs*, unless those are set to "none".

The following update request inserts the data shown above into the repository:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <urn:d4> rdfs:label "This is another literal in English with a language tag for the language only
↪"@en,
            "This is yet another literal tagged for English in Canada"@en-CA,
            "Das ist ein schönes deutsches Literal"@de,
            "Dies hier ist ebenso ein hübsches deutsches Literal, aber aus der Schweiz"@de-CH,
            "96ac1c60-7997-45a3-8dfe-b57b24c1cb62"
}
```

Note: The `"default"` index provides language-agnostic search.

Searching in any of the languages is like in the third example related to *ignoring untagged literals*: the query must specify the index to search.

Searching in the untagged literals can be done like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "b57*"
}
```

Or like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The language tag of the query literal supplies the index to query
    ?value onto:fts "b57*"@default
}
```

Or like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # The query string and the index to query are supplied as two separate values
    # inside an RDF list
```

(continues on next page)

(continued from previous page)

```
    ?value onto:fts ("b57*" "default")
}
```

All of these queries will return the single untagged literal where `"b57*"` was matched to one of the hyphenated components.

	value
1	"96ac1c60-7997-45a3-8dfe-b57b24c1cb62"

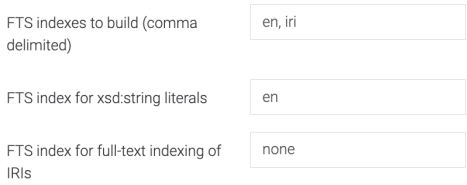
Treat IRIs as keywords and search them

Sometimes, regardless of our need to search literals, we also want to search within IRIs and treat them as keywords so that the entire IRI is considered to be a single searchable token. These can be any IRIs, such as:

- `<http://www.w3.org/2000/01/rdf-schema#domain>`
- `<http://example.com/data/john>`
- `<http://example.com/data/mary>`
- `<http://example.com/data/william>`

To configure the sample search:

1. Create a repository.
2. In its configuration menu, enable a special index called `"iri"` by adding it to the *FTS indexes to build* property. For example, if we also want English literals to be indexed, we will set *FTS indexes to build* to "en, iri".
3. Set *FTS index for xsd:string literals* to "en" so that literals with no language tag will go to the "en" index.



The following update request inserts data that uses the IRIs above:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <http://example.com/data/john> rdfs:label "John" .
    <http://example.com/data/mary> rdfs:label "Mary" .
    <http://example.com/data/william> rdfs:label "William" .
}
```

To search the IRIs, query the `"iri"` index like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # Finds all IRIs that start with "http://example.com/"
    ?value onto:fts "http://example.com/*"@iri
}
```

Or like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # Finds all IRIs that start with "http://example.com/"
```

(continues on next page)

(continued from previous page)

```
    ?value onto:fts ("http://example.com/*" "iri")
}
```

Both of these will return the `http://example.com/xxx` IRIs from the sample data.

	value
1	http://example.com/data/john
2	http://example.com/data/mary
3	http://example.com/data/william

When the entire search string is a single keyword, which is the case for the `"iri"` index, you can also use range searches to find IRIs that fall between two IRIs when sorted:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # Finds all IRIs that sort between http://example.com/data/kelly and http://example.com/data/william,
    # including the boundaries
    ?value onto:fts "[http://example.com/data/kelly TO http://example.com/data/william]"@iri
}
```

Or like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    # Finds all IRIs that sort between http://example.com/data/kelly and http://example.com/data/william,
    # including the boundaries
    ?value onto:fts "[http://example.com/data/kelly TO http://example.com/data/william]"@iri
}
```

Both of these queries should return `http://example.com/data/mary` and `http://example.com/data/william`.

	value
1	http://example.com/data/mary
2	http://example.com/data/william

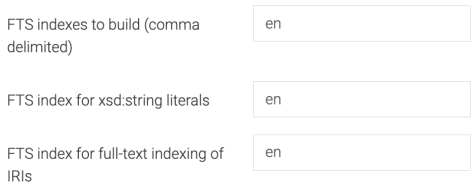
Indexing

In this scenario, regardless of our need to search literals, we also need to search within IRIs, treating them as regular text in which the IRI is split into multiple searchable tokens. These are typically IRIs that are readable and are composed of words:

- `<http://example.com/data/john>`
- `<http://example.com/data/mary>`
- `<http://example.com/data/william>`

To configure the sample search:

1. Create a repository.
2. In its configuration menu, enable the index for the language we want by adding it to *FTS indexes to build* — for English, we set *FTS indexes to build* to “en”.
3. The value of *FTS index for xsd:string literals* must also be set to “en”.
4. We also need IRIs to be indexed for full-text search in the language we enabled, so we set *FTS index for full-text indexing of IRIs* to “en”.



The following update request inserts the data shown above into the repository:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    <http://example.com/data/john> rdfs:label "John" .
    <http://example.com/data/mary> rdfs:label "Mary" .
    <http://example.com/data/william> rdfs:label "William" .
}
```

IRIs are then searchable in the `"en"` index just like literals:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "john"@en
}
```

Or like this:

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts ("john" "en")
}
```

Both of these queries will return the IRI `http://example.com/john`, as well as the literal `"John"`.

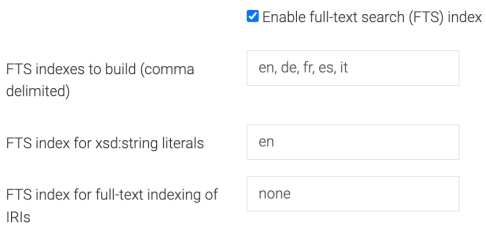
	value
1	"John"
2	http://example.com/data/john

Star Wars dataset examples

These examples use the Star Wars dataset from `starwars-data.ttl`.

Create a repository with the following settings:

- *Enable full-text search (FTS) index*: true
- *FTS indexes to build*: en, de, fr, es, it
- *FTS index for xsd:string literals*: en
- *FTS index for full-text indexing of IRIs*: none (but “en” would also make sense for this dataset)



Let’s look at some example queries.

All literals where “luke” and “vader” are near each other

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts '"luke vader"~5'
}
```

This query returns a single literal.

	value
1	"The Empire Strikes Back (also known as Star Wars: Episode V – The Empire Strikes Back) is a 1980 American epic space opera film directed by Irvin Kershner. Leigh Brackett and Lawrence Kasdan wrote the screenplay, with George Lucas writing the films story and serving as executive producer. The second installment in the original Star Wars trilogy, it was produced by Gary Kurtz for Lucasfilm and stars Mark Hamill, Harrison Ford, Carrie Fisher, Billy Dee Williams, Anthony Daniels, David Prowse, Kenny Baker, Peter Mayhew, and Frank Oz. The film is set three years after Star Wars. The Galactic Empire, under the leadership of the villainous Darth Vader and the Emperor, is in pursuit of Luke Skywalker and the rest of the Rebel Alliance. While Vader relentlessly pursues the small band of Lukes friends—Han Solo, Princess Leia Organa, and others—across the galaxy, Luke studies the Force under Jedi Master Yoda. When Vader captures Lukes friends, Luke must decide whether to complete his training and become a full Jedi Knight or to confront Vader and save them. Following a difficult production, The Empire Strikes Back was released on May 21, 1980. It received mixed reviews from critics initially but has since grown in esteem, becoming the most critically acclaimed chapter in the Star Wars saga; it is now widely regarded as one of the greatest films of all time. The film ranks #3 on Empires 2008 list of the 500 greatest movies of all time. It became the highest-grossing film of 1980 and, to date, has earned more than $538 million worldwide from its original run and several re-releases. When adjusted for inflation, it is the second-highest-grossing sequel of all time and the 13th-highest-grossing film in North America. The film was followed by Return of the Jedi, which was released in 1983. In 2010, the film was selected for preservation in the United States National Film Registry by the Library of Congress for being "culturally, historically, and aesthetically significant.""

Note:

- The query above searches in the `"en"` index since the default index is disabled and we requested xsd:string literals to go to the `"en"` index.
- The query uses single quotes for the query literal to avoid escaping the double quotes that are part of the full-text search query.

All literals containing “skywalker” but not “luke”

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "skywalker -luke"
}
```

This query returns several results, some of which are Luke’s grandmother Shmi Skywalker and Luke’s father Anakin Skywalker before he became Darth Vader.

	value
1	"Anakin Skywalker"@en-GB
2	"Anakin Skywalker"@en
3	"Anakin Skywalker"
4	"Shmi Skywalker"@en
5	"Shmi Skywalker"
6	"Watto is a fictional character in the Star Wars franchise, featured in the films The Phantom Menace and Attack of the Clones. He is computer-generated and played by voice actor Andy Secombe. He is a mean-tempered, greedy Toydarian, and owner of a second-hand goods store in Mos Espa on the planet Tatooine. Among Wattos belongings are the slaves Shmi Skywalker and her son, Anakin. He acquires them after winning a podracing bet with Gardulla the Hutt, and he puts them both to work in his store. Anakin demonstrates an incredible aptitude for equipment repair, and Watto decides to profit from it by having the boy fix various broken equipment in the store. He eventually loses Anakin in a podracing bet with Qui-Gon Jinn when he bets on a competitor, Sebulba, who is defeated by Anakin."

All literals corresponding to a simple FTS query

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "striking jedis"
}
```

This query returns many results, some of which are “The Empire **Strikes** Back” and “Return of the **Jedi**”. This illustrates how full-text search tuned to a specific language (in this case English) is able to match “striking” to “strikes” and “jedis” to “jedi”.

	value
1	"The Empire Strikes Back"
2	"The Empire Strikes Back"@en
3	"The Empire Strikes Back"@en-CA
4	"The Empire Strikes Back"@en-GB
5	"Return of the Jedi"
6	"Return of the Jedi"@en-GB
7	"Return of the Jedi"@en-CA
8	"Return of the Jedi"@en
9	"Jedi starfighter"
10	"Jedi Interceptor"
11	"Star Wars: The Last Jedi"@en
12	"Star Wars: The Last Jedi"@en-CA

Note that the query written like that does not need all tokens to be in the matched result. In other words, the query is equivalent to “striking OR jedis”.

All literals corresponding to a simple FTS query in German

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "das beste"@de
}
```

This query returns matches such as “Ahmed **Best**”, “Oscar für den **besten** Film” and “Oscar für die **beste** Regie”, illustrating the ability of FTS to match different word forms in German.

	value
1	"Ahmed Best"@de
2	"Leanne Best"@de
3	"Leanne Best"@de-AT
4	"Leanne Best"@de-CH
5	"Oscar für den besten Film"@de
6	"Oscar für die beste Regie"@de
7	"Oscar für die beste Hauptdarstellerin"@de

All literals corresponding to a simple FTS query in French

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "oscar acteur"@fr
}
```

This query returns matches like “**Oscar** de la meilleure **actrice**” and “**Oscar** du meilleur **acteur**”, illustrating the ability of FTS to match different word forms in French.

	value
1	"Oscar de la meilleure actrice"@fr
2	"Oscar du meilleur acteur"@fr
3	"acteur"@fr
4	"actrice"@fr
5	"Oscar du meilleur acteur dans un second rôle"@fr
6	"Oscar de la meilleure actrice dans un second rôle"@fr
7	"acteur britannique"@fr

All literals corresponding to a simple FTS query in Italian

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "migliori"@it
}
```

This query returns matches such as “Oscar al **miglior** film”, “Oscar ai **migliori** costumi” and “Oscar alla **migliore** scenografia”, illustrating the ability of FTS to match different word forms in Italian.

	value
1	"Oscar al miglior film"@it
2	"Oscar al miglior regista"@it
3	"Oscar alla miglior attrice"@it
4	"Oscar al miglior attore"@it
5	"Oscar ai migliori costumi"@it
6	"Oscar alla migliore scenografia"@it

All literals corresponding to a simple FTS query in Spanish

```
PREFIX onto: <http://www.ontotext.com/>
select * {
    ?value onto:fts "peliculas"@es
}
```

This query returns matches like “**Película** del 2005” and “personaje de ficción el las **películas** de Star Wars”, illustrating the ability of FTS to match different word forms in Spanish but also the ability to ignore diacritics when searching.

	value
1	"película de 1999"@es
2	"Película del 2005"@es
3	"Óscar a la mejor película"@es
4	"Película de space opera de 1977"@es
5	"película de la saga Star Wars"@es
6	"película estadounidense de ciencia ficción"@es
7	"Óscar a la mejor película de animación"@es
8	"personaje de ficción el las películas de Star Wars"@es

3.4.5 SPARQL Federation

SPARQL 1.1 Federation provides extensions to the query syntax to support the execution of distributed queries over any number of SPARQL endpoints. This powerful feature enables integration of RDF data from different sources using a single query.

For example, to discover DBpedia resources about people who have the same names as those stored in a local repository, you could use the following query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT ?dbpedia_id
WHERE {
   ?person a foaf:Person ;
           foaf:name ?name .
   SERVICE <https://dbpedia.org/sparql> {
```

(continues on next page)

(continued from previous page)

```
        ?dbpedia_id a dbpedia-owl:Person ;
                    foaf:name ?name .
    }
}
```

It matches the first part against the local repository, and for each person it finds, it checks the DBpedia SPARQL endpoint to see if a person with the same name exists. If so, it returns the ID.

Note: Use caution with federation. First, to avoid excessive querying of remote public SPARQL endpoints, but also because it can lead to inefficient query patterns.

The following example finds resources in the second SPARQL endpoint that have an `rdfs:label` value that is similar to the `rdfs:label` value of `<http://dbpedia.org/resource/Vaccination>` in the first SPARQL endpoint:

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
SELECT ?endpoint2_id {
    VALUES ?endpoint1_id {
        <http://dbpedia.org/resource/Vaccination>
    }
    SERVICE <http://faraway_endpoint.org/sparql> {
        ?endpoint1_id rdfs:label ?l1 .
        FILTER( langMatches(lang(?l1), "en") )
    }
    SERVICE <http://remote_endpoint.com/sparql> {
        ?endpoint2_id rdfs:label ?l2 .
        FILTER( str(?l2) = str(?l1) )
    }
}
```

However, such a query is very inefficient, because no intermediate bindings are passed between endpoints. Instead, both subqueries execute independently, requiring the second subquery to return all `X rdfs:label Y` statements that it stores. These are then joined locally to the (likely much smaller) results of the first subquery.

You can optimize query execution by batching multiple values. Keep in mind that:

- The default batching size is `15`, which is OK to use in most cases.
- You can change the default via the `graphdb.federation.block.join.size` global property.
- By using a system graph, you can set a value only for a particular query evaluation.

3.4.5.1 Internal SPARQL federation

Since RDF4J repositories are also *SPARQL endpoints*, you can use the federation mechanism to do distributed querying over several repositories on a local server. You can do this by referring to them as a standard `SERVICE` with their full path, or, if they are running on the same GraphDB instance, you can use the optimized local `repository` prefix. This prefix triggers the internal federation mechanism. The internal SPARQL federation is used in almost the same way as the standard SPARQL federation over HTTP, and has several advantages:

Speed The HTTP transport layer is bypassed and iterators are accessed directly. The speed is comparable to accessing data in the same repository.

Security When security is ON, you can access every repository that is readable by the currently authenticated user. (Standard SPARQL 1.1 federation does not support authentication.)

Flexibility Inline parameters provide control over inference and statement expansion over `owl:sameAs`.

Usage

Instead of a URL to a remote repository, provide a special URL of the form `repository:NNN`, where `NNN` is the ID of the repository you want to access. For example, you could access the `authors` repository via internal federation with a query like this:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {
    ?book rdfs:label "The Hitchhiker's Guide to the Galaxy" ;
        books:author ?author .

    SERVICE <repository:authors> {
        ?author rdfs:label ?authorName
    }
}
```

The approach used in the DBpedia example above, `SERVICE <http://localhost:7200/repositories/my_labels>`, is also valid but is less efficient.

Parameters

There are four parameters that control how the federated part of the query is executed:

Parameter	Definition
`infer` (boolean)	Controls whether inferred statements are included. `True` by default. When set to `false`, it is equivalent to adding `FROM <http://www.ontotext.com/explicit>` to the federated query.
`sameAs` (boolean)	Controls whether statements are expanded over `owl:sameAs`. `True` by default. When set to `false`, it is equivalent to adding `FROM <http://www.ontotext.com/disable-sameAs>` to the federated query.
`from` (string)	Can be repeated multiple times; translates to `FROM <...>`. No default value.
`fromNamed` (string)	Can be repeated multiple times; translates to `FROM NAMED <...>`. No default value.

To set a parameter, put a comma after the special URL referring to the internal repository, then the parameter name, an equals sign, and finally the value of the parameter. If you need to set more than one parameter, put another comma, parameter name, equals sign, and value.

Some examples:

`repository:NNN,infer=false` Turns off inference and inferred statements are not included in the results.

`repository:NNN,sameAs=false` Turns off the expansion of statements over `owl:sameAs` and they are not included in the results.

`repository:NNN,infer=false,sameAs=false` Turns off the inferred statements and they are not included in the results.

Turns off the expansion of statements over `owl:sameAs` and they are not included in the results.

`service <repository:repo1>` No `FROM` and `FROM NAMED`.

`service <repository:repo1,from=http://test.com>` Adds `FROM <http://test.com>`.

`service <repository:repo1,fromNamed=http://test.com/named>` Adds `FROM NAMED <http://test.com/named>`.

`service <repository:repo1,from=http://test.com,fromNamed=http://test.com/named,sameAs=false>` Adds `FROM <http://test.com>`, adds `FROM NAMED <http://test.com/named>`, and does not expand over `owl:sameAs`.

Note: This needs to be a valid URL, so so you cannot use spaces.

The example SPARQL query from above will look like this if you want to skip the inferred statements and disable the expansion over `owl:sameAs`:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {
    ?book rdfs:label "The Hitchhiker's Guide to the Galaxy" ;
        books:author ?author .

    SERVICE <repository:authors,infer=false,sameAs=false> {
        ?author rdfs:label ?authorName
    }
}
```

3.4.5.2 Federated query to a remote password-protected repository

You can also use federation to query a remote password-protected GraphDB repository or a SPARQL endpoint by *adding the other GraphDB instance as a remote location* and specifying the credentials for it.

Once enabled, you can query a GraphDB remote location on `graphdb2.example.com:7200` as follows, where `example-repo` is the ID of the remote repository:

```
PREFIX ex: <http://example.com/>
SELECT ?id ?label
WHERE {
    ?id a ex:Concept .
    SERVICE <http://graphdb2.example.com:7200/repositories/example-repo> {
        ?id rdfs:label ?label.
    }
}
```

You can use any URL parameters supported by the remote endpoint. For example, if the remote repository is an RDF4J or GraphDB repository, you can query it by using the URL `http://factforge.net/repositories/ff-news?infer=false` to include only explicit statements.

You can query a remote SPARQL endpoint as follows:

```
PREFIX ex: <http://example.com/>
SELECT ?id ?label
WHERE {
    ?id a ex:Concept .
    SERVICE <repository:my_labels> {
        ?id rdfs:label ?label.
    }
}
```

3.4.6 The SPARQL-MM multimedia extension

SPARQL-MM is a multimedia extension for SPARQL 1.1 that makes it easier to query media metadata about temporal relations, spatial relations, and more. The implementation is based on code developed by Thomas Kurz, and is implemented as a GraphDB plugin.

Tip: For more information, see:

- The SPARQL-MM Specification
- SPARQL-MM — Extending SPARQL to Media Fragments (Springer)
- List of SPARQL-MM functions

3.4.6.1 Usage examples

Temporal relations

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm:precedes(?f1,?f2)
} ORDER BY ?t1 ?t2
```

Temporal aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm:  <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:temporalIntermediate(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

Temporal accessors

```
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 WHERE {
    ?f1 a ma:MediaFragment.
} ORDER BY mm:duration(?f1)
```

Spatial relations

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm:rightBeside(?f1,?f2)
} ORDER BY ?t1 ?t2
```

Spatial aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm:  <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:spatialIntersection(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

General relation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm:equals(?f1,?f2)
} ORDER BY ?t1 ?t2
```

General aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm:  <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:boundingBox(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

General accessor

```
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?pixelURI WHERE {
    ?f1 ma:hasFragment ?f1.
    BIND (mm:toPixel(?f1) AS ?pixelURI)
} ORDER BY ?t1 ?t2
```

3.4.7 Data history and versioning

The Data history and versioning plugin lets you access past states of your database through versioning of the RDF data model level. Collecting and querying the history of a database helps users and organizations that want to preserve all of their historical data and who often need to know: when did this value in the database change and what was the previous system state?

The plugin remembers changes from multiple transactions and provides the means to track historical data. Changes in the repository are tracked globally for all users, and all updates can be queried and processed at once. The tracked data is persisted to disk and is available after a restart.

This can be useful in use cases such as:

- Generating a "diff" between generations while data updates are loaded into the system on a regular basis, either through ETL or a change data stream.
- Showing what has changed between moment A and moment B, for example: "After an application change was implemented over the weekend, I need to compare the deployment footprint or configuration from before and after the application change".
- Maintaining history only for specific classes or properties with no need for keeping the history of everything. This provides a significant advantage when working with very large databases, the querying of which would require substantial amounts of time and system resources.
- Searching for the members of a specific team at point X.

Warning: Note that querying the history log may be slow for big history logs. This is why we recommend using *filters* to reduce the number of history entries if you have a big repository.

3.4.7.1 Usage

Enable or disable a plugin

Enabling and disabling the plugin refers to collecting history only, and is disabled by default. Querying the collected history is possible at any moment.

To enable the plugin, execute the following update request:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/enabled> true
}
```

To disable it, execute:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/enabled> false
}
```

To check the current enabled status, execute:

```
SELECT ?enabled {
    [] <http://www.ontotext.com/at/enabled> ?enabled
}
```

Transaction IDs

As shown in *this query example*, each transaction can be identified both by its timestamp as well as a transaction ID. The transaction ID is an IRI and by default is derived from the timestamp.

Setting your own transaction IDs

You can assign a persistent IRI of your choosing as the transaction ID for a given transaction, and it will be used as the subject of all triples that record this transaction in the data history. This makes the querying and other management of transaction data much easier.

The transaction ID must be a valid IRI and it must be one that has not already been used as a transaction ID in the collected history.

The following example demonstrates the use of the `hist:transactionId` predicate to assign the URI `http://example.com/t14` via a SPARQL update that replaces a triple about a resource's given name value with another triple that has a new value for that name:

```
PREFIX hist: <http://www.ontotext.com/at/>

INSERT DATA {
    [] hist:transactionId <http://example.com/t14> .
};

DELETE DATA {
    <urn:Kirk> <urn:givenName> "Jim" .
};

INSERT DATA {
    <urn:Kirk> <urn:givenName> "James" .
}
```

The statement that sets the transaction ID must be the first change in the transaction.

Clear all data

Before clearing all data in your repository, you should first disable collecting history, as there is no way to have usable history after this operation has been executed. For example:

- You try to execute `CLEAR ALL`, but get an error: The reason is that clearing all statements in the repository is incompatible with collecting history. Disable collecting history if you really want to clear all data.
- You disable collecting history and retry `CLEAR ALL`: All data in the repository is deleted. All history data is deleted as well, since whatever is there is no longer usable.

Clear history

You can also delete just the history without deleting the data in the repository or disabling the collection of history, as shown here:

```
PREFIX hist: <http://www.ontotext.com/at/>
INSERT DATA {
    [] hist:clearHistory [] .
}
```

Trim history

Special predicates are available to let you trim the history based on various criteria.

Delete history before a certain date

```
PREFIX hist: <http://www.ontotext.com/at/>
INSERT DATA {
    [] hist:trimBefore "2022-11-29"  .
}
```

The provided literal must be a valid `xsd:date` or `xsd:dateTime` value. If only the date is specified, GraphDB assumes the time to be midnight (00:00:00). The timezone is by default the system timezone. For more precise trimming, a full datetime value should be specified.

Trim history by size

Size here means the number of statements in the history log to be preserved.

```
PREFIX hist: <http://www.ontotext.com/at/>
INSERT DATA {
    [] hist:trimToSize 1000 .
}
```

Trim the history to a given period from the current date and time

```
PREFIX hist: <http://www.ontotext.com/at/>
INSERT DATA {
    [] hist:trimToPeriod "P3D" .
}
```

The provided literal must be a valid `xsd:duration` value. P3D here means 3 days, so only the history from the last 3 days would remain after executing the update. We can also specify the duration in minutes or hours.

History filtering

Because keeping history for everything is usually unnecessary, as well as quite time and resource consuming, this plugin lets you specify only certain classes or properties whose history should be recorded. When configuring the index, you need to specify 4 mandatory positions — subject, predicate, object, and context. Each position can have one of the following values:

- `*`: Everything is allowed.
- `!(IRI, Bnode, or Literal)`: Anything apart from the selected type is allowed.
- `IRI`, `BNode` or `Literal`: The type of the entity on this position must be the specified one. Names are not case-sensitive.
- an IRI: Only this IRI is allowed.
- an IRI prefix (`http://myIRI*`): All IRIs that start with the given prefix are allowed.

Filter examples

- `* * literal *`: Match statements that contain any literal in the object position.
- `* * !literal *`: Match statements that do not contain any literal in the object position.
- `* http://example.com/name * *`: Match statements whose predicate is `http://example.com/name`.
- `http://example.com/person/* * * *`: Match statements whose subject is an IRI starting with `http://example.com/person/`.

A statement is kept in the history if it matches **at least one** of the provided statement templates.

Manage filters

- Add filter:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/addFilters> "* * LITERAL *"
}
```

- Remove filter:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/removeFilters> "* * LITERAL *"
}
```

- List filters:

```
SELECT ?filter WHERE {
    [] <http://www.ontotext.com/at/getFilters> ?filter
}
```

3.4.7.2 Query process and examples

1. Enable the plugin:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/enabled> true
}
```

2. Insert the data you want to query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT DATA {
    <urn:Human> rdfs:subClassOf <urn:Mammal> .
    <urn:Commander> rdfs:subClassOf <urn:StarfleetOfficer> .
    <urn:Captain> rdfs:subClassOf <urn:StarfleetOfficer> .
    <urn:Kirk> a <urn:Human> ;
        <urn:dateOfBirth> "2233-03-22"^^xsd:date ;
        <urn:name> "James T. Kirk" ;
        <urn:rank> <urn:Commander> .
}
```

3. Change the name of a particular Starfleet officer, so that you can then see how this change is tracked:

```
DELETE DATA {
    <urn:Kirk> <urn:name> "James T. Kirk"
};
```

(continues on next page)

(continued from previous page)

```
INSERT DATA {
    <urn:Kirk> <urn:name> "James Tiberius Kirk"
}
```

4. Query the history of your data:

   a. Find out the **specific point in time** when data was changed by browsing the history with the following query:

```
PREFIX hist: <http://www.ontotext.com/at/>
SELECT * {
    ?transactionId a hist:history ;
        hist:timestamp ?time ;
        hist:graph ?g ;
        hist:subject ?s ;
        hist:predicate ?p ;
        hist:object ?o ;
        hist:insert ?i ;
        hist:username ?username .
}
```

The retrieved results are in reverse chronological order:

	transactionId	time	g	s	p	o	i	username
1	http://www.ontotext.com/at/20231012082429161	"2023-10-12T11:24:29.161+03:00"^^xsd:dateTime		urn:Kirk	urn:name	"James T. Kirk"	"false"^^xsd:boolean	"mary.green"
2	http://www.ontotext.com/at/20231012082429161	"2023-10-12T11:24:29.161+03:00"^^xsd:dateTime		urn:Kirk	urn:name	"James Tiberius Kirk"	"true"^^xsd:boolean	"mary.green"
3	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Human	rdfs:subClassOf	urn:Mammal	"true"^^xsd:boolean	"john.smith"
4	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Commander	rdfs:subClassOf	urn:StarfleetOfficer	"true"^^xsd:boolean	"john.smith"
5	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Captain	rdfs:subClassOf	urn:StarfleetOfficer	"true"^^xsd:boolean	"john.smith"
6	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Kirk	rdf:type	urn:Human	"true"^^xsd:boolean	"john.smith"
7	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime	http://www.ontotext.com/implicit	urn:Kirk	rdf:type	urn:Mammal	"true"^^xsd:boolean	"john.smith"
8	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Kirk	urn:dateOfBirth	"2233-03-22"^^xsd:date	"true"^^xsd:boolean	"john.smith"
9	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Kirk	urn:name	"James T. Kirk"	"true"^^xsd:boolean	"john.smith"
10	http://www.ontotext.com/at/20231012082330759	"2023-10-12T11:23:30.759+03:00"^^xsd:dateTime		urn:Kirk	urn:rank	urn:Commander	"true"^^xsd:boolean	"john.smith"

The subject of the `?transactionId a hist:historyId` will be bound to the *transaction ID* of each transaction in the collected history.

The username of the user who executed the transaction can be accessed via the `hist:username` predicate. In this example, the initial data was imported by "john.smith", while the last change was executed by "mary.green".

   b. Let's see how we can use a **negation filter**.

      i. Run the following update request to apply the filter shown above that states that no literal can be in the object position:

```
INSERT DATA {
    [] <http://www.ontotext.com/at/addFilters> "* * !LITERAL *"
}
```

      ii. Now, let's add a second date of birth for the Commander:

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT DATA {
    <urn:Kirk> <urn:dateOfBirth> "2633-03-22"^^xsd:date .
}
```

iii. Re-running the query from 4.a will show that the data has not been added since it is a literal.

c. You can also find out **what changes were made for a subject and a predicate within a specific time period**. This is done with the `hist:listWithFilter` magic predicate:

```
[] hist:listWithFilter (?fromDateTime ?toDateTime ?subject ?predicate ?
↪object ?context).
```

Note: The `hist:listWithFilter` magic predicate replaces the `hist:parameters` one, which worked the same way but is deprecated as of release 10.4. Note that `hist:listWithFilter` is used directly and not together with `?transactionId a hist:history`.

Because of this, `hist:listWithFilter` will not bind the transaction ID in the subject position. If you need the transaction ID, you can access it through the `hist:transactionId` predicate as illustrated in the examples below.

While the predicate is not mandatory, passing parameters when querying history data is much more efficient than fetching all history elements and then filtering them. Keep in mind that their order is important, and when present, the predicate will only return history entries that match the list. Only bound variables will be taken, and there may also be unbound parameters. Not all bindings are required, but since the object list is an ordered list, if you want to filter by subject for example, you must add at least `?fromDateTime ?toDateTime ?subject` as bindings. `?fromDateTime ?toDateTime` may be left unbound.

The following query returns **all changes made within a given time period**:

```
PREFIX hist: <http://www.ontotext.com/at/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT * {
    [] hist:listWithFilter ("2022-07-12T16:17:00"^^xsd:dateTime "2022-07-
↪12T16:20:00"^^xsd:dateTime);
       hist:transactionId ?transactionId ;
       hist:timestamp ?time ;
       hist:graph ?g ;
       hist:subject ?s ;
       hist:predicate ?p ;
       hist:object ?o ;
       hist:insert ?i
}
```

You can also query for **all changes for a particular subject and predicate**. Note that the `?fromDateTime ?toDateTime` parameters are left unbound.

```
PREFIX hist: <http://www.ontotext.com/at/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?time ?s ?p ?o ?i {
    [] hist:listWithFilter (?fromDateTime ?toDateTime <urn:Kirk> <urn:name>
↪?object ?context);
       hist:transactionId ?transactionId ;
       hist:timestamp ?time ;
       hist:graph ?g ;
       hist:subject ?s ;
       hist:predicate ?p ;
       hist:object ?o ;
       hist:insert ?i
}
```

Filter query results — Showing results from 1 to 3 of 3. Query took 0.1s, moments ago.

	time	s	p	o	i
1	"2022-07-12T16:17:17.354+03:00"^^xsd:dateTime	urn:Kirk	urn:name	"James T. Kirk"	"false"^^xsd:boolean
2	"2022-07-12T16:17:17.354+03:00"^^xsd:dateTime	urn:Kirk	urn:name	"James Tiberius Kirk"	"true"^^xsd:boolean
3	"2022-07-12T16:17:06.303+03:00"^^xsd:dateTime	urn:Kirk	urn:name	"James T. Kirk"	"true"^^xsd:boolean

d. You can **query the data at a specific point in time** by including `FROM <http://www.ontotext.com/at/xxx>`, where `xxx` is a date-time in the format: *yyyy[[[[MM]dd]HH]mm]ss]*. For example:

```
# Return data as it looked on 2022-07-12 16:17:17 server time
#
SELECT ?name ?rank ?dateOfBirth FROM <http://www.ontotext.com/at/
↪20220712161717> {
    BIND(<urn:Kirk> as ?officer)
    ?officer <urn:name> ?name ;
            <urn:rank> ?rank ;
            <urn:dateOfBirth> ?dateOfBirth .
}
```

Filter query results — Showing results from 1 to 1 of 1. Query took 0.3s, moments ago.

	name	rank	dateOfBirth
1	"James Tiberius Kirk"	urn:Commander	"2233-03-22"^^xsd:date

The same query will return a valid graph with only the date specified:

```
# Return data as it looked on 2022-07-12 00:00:00 server time
# (explicit year and month only)
#
SELECT ?name ?rank ?dateOfBirth FROM <http://www.ontotext.com/at/20220712> {
    BIND(<urn:Kirk> as ?officer)
    ?officer <urn:name> ?name ;
            <urn:rank> ?rank ;
            <urn:dateOfBirth> ?dateOfBirth .
}
```

To retrieve all data for that particular Starfleet officer at a specific point in time, you can also use a `DESCRIBE` query:

```
DESCRIBE <urn:Kirk> FROM <http://www.ontotext.com/at/20220712161717>
```

The result from our example at that point in time would be:

Filter query results — Showing results from 1 to 5 of 5. Query took 0.1s, moments ago.

	subject	predicate	object
1	urn:Kirk	rdf:type	urn:Human
2	urn:Kirk	rdf:type	urn:Mammal
3	urn:Kirk	urn:dateOfBirth	"2233-03-22"^^xsd:date
4	urn:Kirk	urn:name	"James T. Kirk"
5	urn:Kirk	urn:rank	urn:Commander

Note: Statements that have history will use the history data according to the requested point in time. Statements that do not have history will be returned directly, assuming they were never modified and existed at the requested point as well.

3.4.7.3 Index components

The plugin index is of the type DSPOCI, meaning that it consists of the following components indexed in the order shown:

- Date-time — a 64-bit long value that represents the exact time an operation occurred with millisecond precision. All operations in the same transaction have the same date-time value.
- Subject — the statement subject, 32 or 40 bit long.
- Predicate — the statement predicate, 32 or 40 bit long.
- Object — the statement object, 32 or 40 bit long.
- Context — the statement context, 32 or 40 bit long. Special values are used for explicit statements in the default graph and for implicit statements. By including the implicit statements, we get transparent support for transactions.
- Insert — a boolean value stored with the fewest bits that make sense. `True` represents an `INSERT`, and `false` represents a `DELETE`.

The index is ordered by each component going from left to right, where the date-time component is in reverse chronological order, and all other components are ordered in ascending order. For example:

Date-time	Subject	Predicate	Object	Context	Insert
`1570623056397`	`urn:Kirk`	`urn:name`	`"James Tiberius. Kirk"`	`<http://www.ontotext.com/explicit>`	`False`
`1570623056397`	`urn:Kirk`	`urn:name`	`"James. T. Kirk"`	`<http://www.ontotext.com/explicit>`	`True`

Tip: Due to the order of the index components, the most time-efficient way to query your data is first by date-time and then by subject. This is particularly valid when *listing with a filter*.

3.4.8 SQL access over JDBC

As a data scientist or an engineer with experience in specific SQL-based tools, you might want to consume RDF data from your knowledge graph or other RDF databases by accessing GraphDB via a Business Intelligence (BI) tool such as Tableau or Microsoft Power BI. GraphDB's JDBC driver makes this possible by letting you use SPARQL `SELECT` queries (with access to all GraphDB features including plugins and SPARQL federation) to create SQL views. This support is based on the Apache Calcite protocol and on performing optimizations and mappings.

The JDBC driver works with preconfigured SQL views (tables) that are saved under each repository whose data we want to access. To simplify the table creation process, we have integrated the SQL View Manager in the GraphDB Workbench. This lets you configure, store, update, preview, and delete SQL views that can be used with the JDBC driver, where each SQL view is based on a SPARQL `SELECT` query and requires additional metadata in order to configure the SQL columns.

Important: This feature only provides read access to a repository. Write operations are not enabled.

3.4.8.1 Configuration

Prerequisites

You need to download the GraphDB JDBC driver (graphdb-jdbc-remote-11.3.3.jar), a self-contained `.jar` file. Install the driver according to the requirements of the software (for example, a BI tool) that will be communicating with GraphDB via JDBC.The driver needs to be installed according to the requirements of the software that supports JDBC. See below for specific instructions.

The sample configuration described here will use the `Netherlands restaurants` RDF dataset. Upload it into a new GraphDB repository, name it "nl_restaurants", and set it as the active repository.

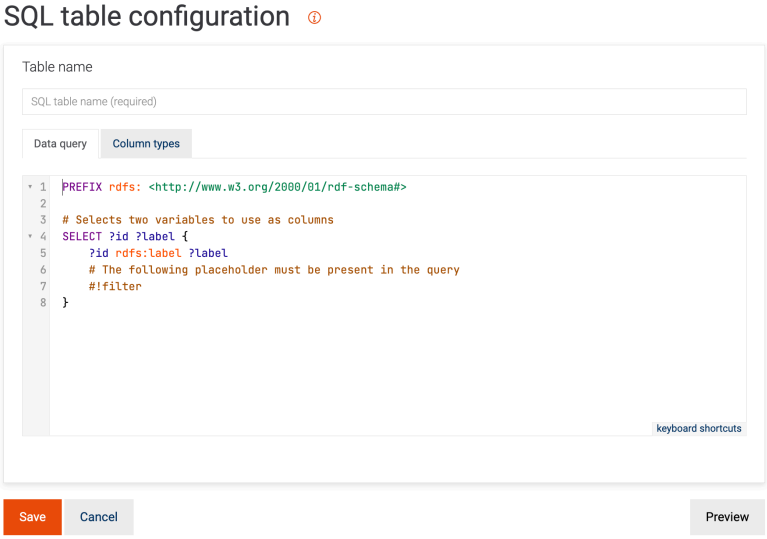
The next step is to create an SQL view of this data.

Creating a SQL view

1. Go to *Setup* ▸ *JDBC*. Initially, the list of SQL table configurations will be empty if none are configured.
2. Click *Create new SQL table configuration*.

   In the view that opens, there are two tabs:

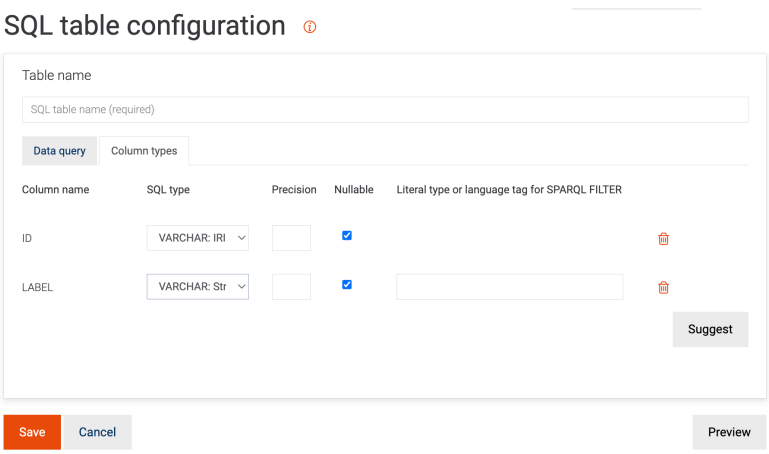
   - *Data query*: The editor where you specify the SPARQL `SELECT` query that is abstracted as a SQL view for the JDBC driver. By default, it opens with a simple SPARQL query that defines an `id` column and a `label` column that uses the `rdfs:label` value.



> **Note:** The query contains a special comment in the query body that specifies the position of the `filter` clause that will be generated on the SQL side. Make sure that it is spelled out **in lowercase**, as otherwise the query parser will not recognize it.

   - *Column types*: The SQL column *types* and other metadata of the SQL table. Hover over a field or a checkbox to see more information about it in a tooltip.

     A table must have at least one column.

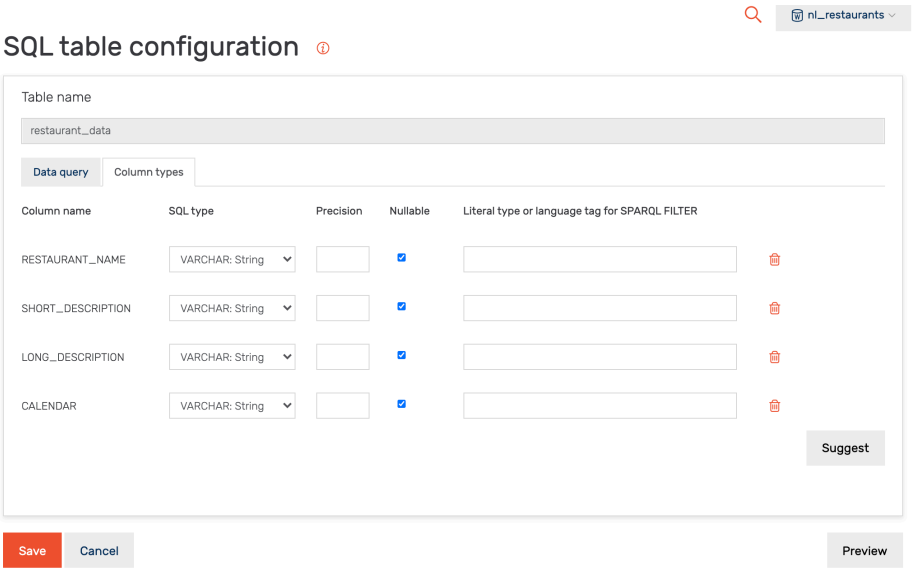


3. Fill in a *Table name* such as "restaurant_data". This mandatory field cannot be changed once the table has been created.
4. Now, let's edit the SPARQL `SELECT` query in the *Data query* body.

   Enter the following query in the editor:

```
PREFIX ex:<http://example.com/ex>
PREFIX base:<http://example/base/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

select ?restaurant_name ?short_description ?long_description ?calendar where {
    ?s a base:Restaurant;
        rdfs:label ?restaurant_name;
    ex:shortDescription ?short_description;
    ex:longDescription ?long_description;
    ex:calendar ?calendar.
    # !filter
}
```

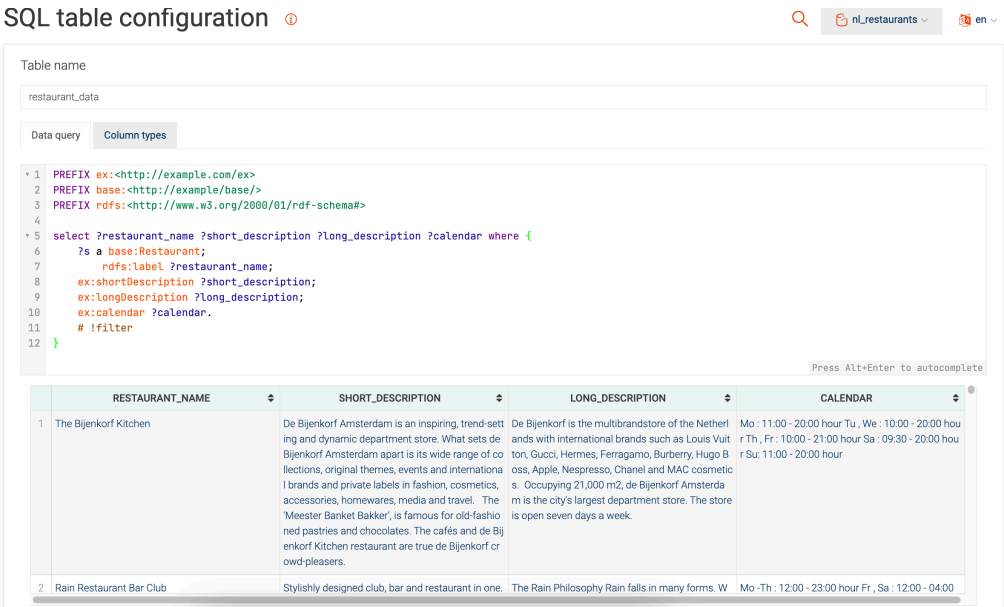
5. After adding the SPARQL `SELECT` query, go to the *Column types* tab and click the *Suggest* button. This will generate all possible columns based on the bindings inside the `SELECT` query. It will also suggest SQL types based on the xsd types from the first 100 results of the execution of the input query:



6. Here, you can:
   - Update the *SQL type* of each column. This is the only mandatory field.

- Configure the precision of the SQL type if applicable (for example, with decimal values).
- Make a column NOT NULL (default is *Nullable*).
- Provide a *Literal type or language tag for SPARQL FILTER*.

7. You can also remove a column from the configuration with the delete icon on the right. If you want to add it again later, you can do so with the *Suggest* button, which will automatically add it again and suggest types for the columns.
8. After configuring the table columns, return to the *Data query* tab and *Preview* the table that it would return. You do not need to save it before previewing it.



Note: If you click *Cancel* before saving, a warning will notify you that you have unsaved changes.

9. After successfully configuring the SQL view, we can *Save* it. It will appear in the list of configured tables that can be used with the JDBC driver.

For the purposes of the BI tool examples in the *Usage examples* section further below, let's also create another SQL view with the following query:

```
PREFIX ex:<http://example.com/ex>
PREFIX base:<http://example/base/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

select ?restaurant_name ?city ?country ?address ?zipcode ?latitude ?longitude where {
    ?s a base:Restaurant;
        rdfs:label ?restaurant_name;
        ex:inCity ?city_id;
        ex:address ?address;
        ex:zipcode ?zipcode;
        ex:latitude ?latitude;
        ex:longitude ?longitude.

    ?city_id rdfs:label ?city.
    ?city_id ex:in ?country_id.
    ?country_id rdfs:label ?country.
    # !filter
}
```

Name it "restaurant_location" and save it.

Updating a SQL view

To edit and update a SQL view, select it from the list of available SQL views that are configured for the selected repository. The configuration is identical to that used for creation, except that here you cannot update the name of the SQL view. You can edit and update the query and SQL column metadata.

After updating the configuration, you can *Save* and see that all changes have been reflected.

Deleting a SQL view

To delete a SQL view, click the delete icon next to its name in the available SQL views list.

3.4.8.2 Type mapping

This table shows all RDF data types, their type equivalent in SQL, and the conversion (or mapping) of RDF to SQL values.

Metadata type	SQL type	Default precision and scale	RDF to SQL	Default RDF type in FILTER()
string	VARCHAR	1,000	Literal.stringValue()	plain literal or literal with language tag
IRI	VARCHAR	500	IRI.stringValue()	IRI
boolean	BOOLEAN		Literal.booleanValue()	literal with xsd:boolean
byte	BYTE		Literal.byteValue()	literal with xsd:byte
short	SHORT		Literal.shortValue()	literal with xsd:short
int	INT		Literal.intValue()	literal with xsd:int
long	LONG		Literal.longValue()	literal with xsd:long
float	FLOAT		Literal.floatValue()	literal with xsd:float
double	DOUBLE		Literal.doubleValue()	literal with xsd:double
decimal	DECIMAL	19, 0	Literal.decimalValue()	literal with xsd:decimal
date	DATE		See below	literal with xsd:date, no timezone
time	TIME		See below	literal with xsd:time, no timezone
timestamp	TIMESTAMP		See below	literal with xsd:datetime, no timezone

Each metadata type may be followed by optional precision and scale in parentheses, such as decimal(15,2) or string(100), and an optional nullability specification that consists of the literal null or not null. By default, all columns are nullable.

RDF values are converted to SQL values on a best effort basis. For example, if something was specified as “long” in SQL, it will convert to a long value if the corresponding literal looks like a long number regardless of its datatype. If the conversion fails (for example, if “foo” cannot be parsed as a long value), the SQL value will become null.

The default RDF type is used only to construct values when a condition from an SQL `WHERE` clause is pushed to a SPARQL `FILTER()`.

Dates, times, and timestamps are tricky because of SQL’s lack of timezone support. There are SQL types with timezone support but they are not implemented fully in Calcite. In order to provide a most common use case, we proceed as follows:

- Ignore the timezone on date and time literals.

  Dates such as 2020-07-01, 2020-07-01Z, 2020-07-01+03:00, and 2020-07-01-03:00 will all be converted to 2020-07-01.

  Times such as 12:00:01, 12:00:01Z, 12:00:01+03:00, and 12:00:01-03:00 will all be converted to 12:00:01.

  No timezone will be added when constructing a value for filtering.

- On datetime values GraphDB considers "no timezone" to be equivalent to "Z" (+00:00). All other timezones will be converted by adjusting the datetime value by the respective offset.

  No time zone will be added when constructing a value for filtering.

3.4.8.3 WHERE to FILTER conversion

The following SQL operators are converted to `FILTER` conditions and pushed to SPARQL, if possible:

- Equality: `=`, `<>`, `<`, `<=`, `>=`
- Nullability: `IS NULL`, `IS NOT NULL`
- Text search: `LIKE`, `SIMILAR TO`

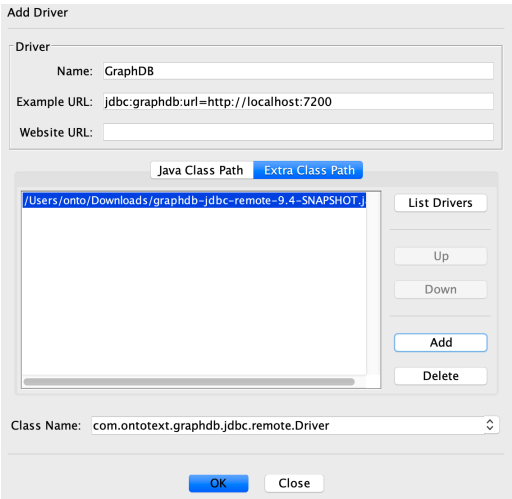
The conversion happens only if one of the operands is a column and the other one is a constant.

3.4.8.4 Table verification

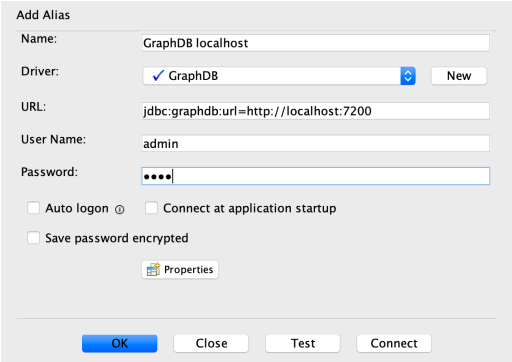
We can also use an external tool such as SQuirrel Universal SQL Client to verify that the SQL table that we created through Workbench is functioning properly.

After installing it, execute the following steps:

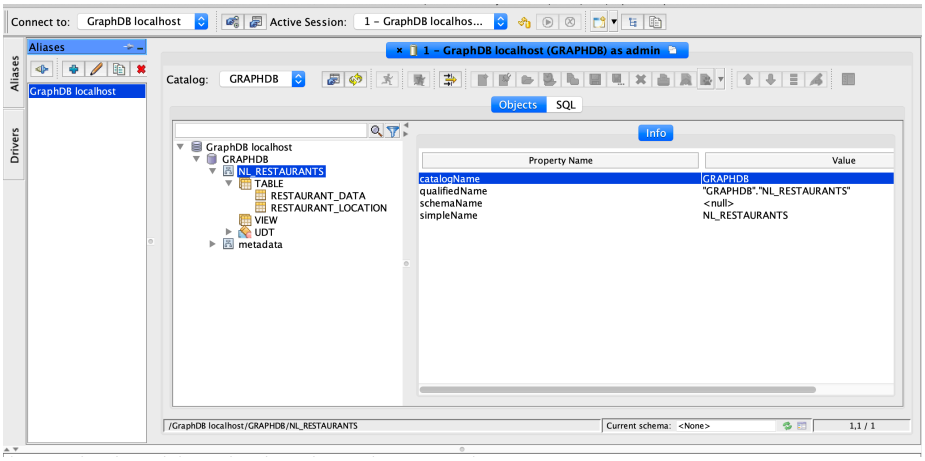
1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-11.3.3.jar), a self-contained `.jar` file.
2. Open SQuirrel and add the JDBC driver: go to the *Drivers* tab on the left and click the + icon to create a new driver.
3. In the dialog box, select *Extra Class Path* and click *Add*.
4. Go to the driver's location on your computer, select it, and click *Choose*.
5. In the *Name* field, choose a name for the driver such as "GraphDB".
6. For *Example URL*, enter the string `jdbc:graphdb:url=http://localhost:7200` (or the respective endpoint URL if your repository is in a remote location).
7. For *Class Name*, enter `com.ontotext.graphdb.jdbc.remote.Driver`. Click *OK*.



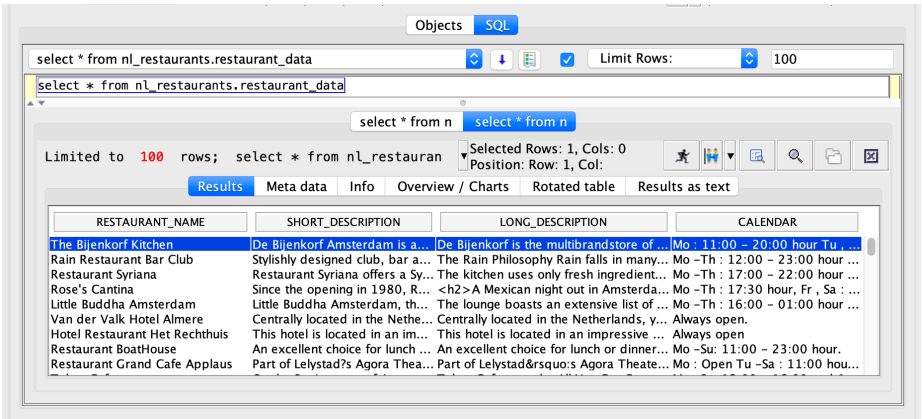
8. Now go to the *Aliases* tab on the left and click + to create a new one.
9. You will see the newly created driver and its URL visible in the dialog box. Choose a name for the alias such as “GraphDB localhost”. Username “admin” and password “root” are only necessary if GraphDB security is enabled.



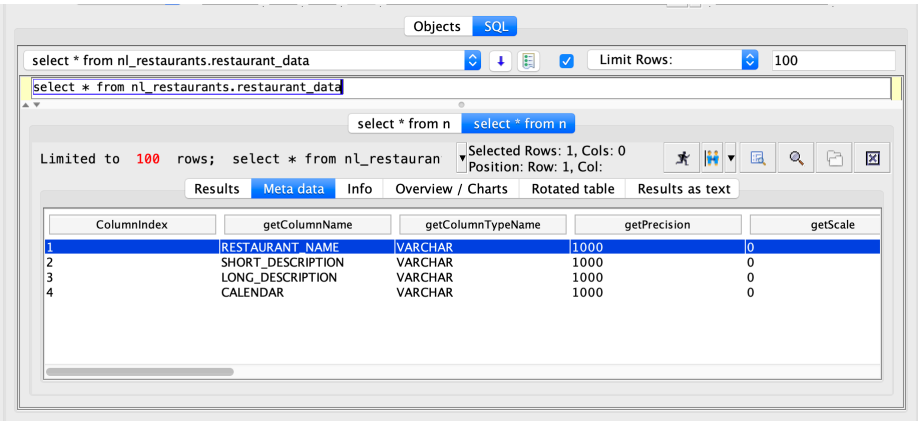
10. You can now see your repository with the two tables that it contains:



11. In the *SQL* tab, you can see information about the tables such as their content. Write your SQL query in the empty field and press Ctrl+Enter (or the *Run SQL* icon above):



You can also see the metadata:

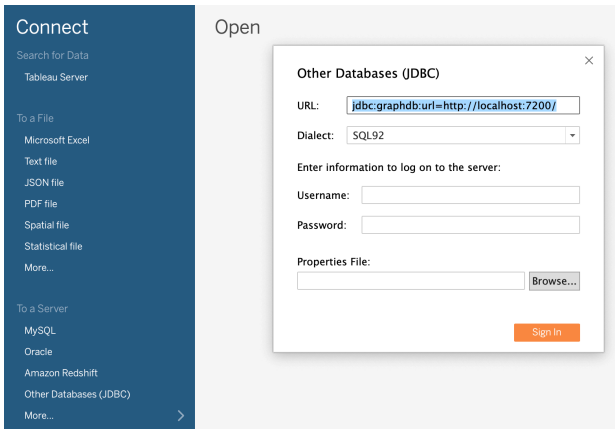


3.4.8.5 Usage examples

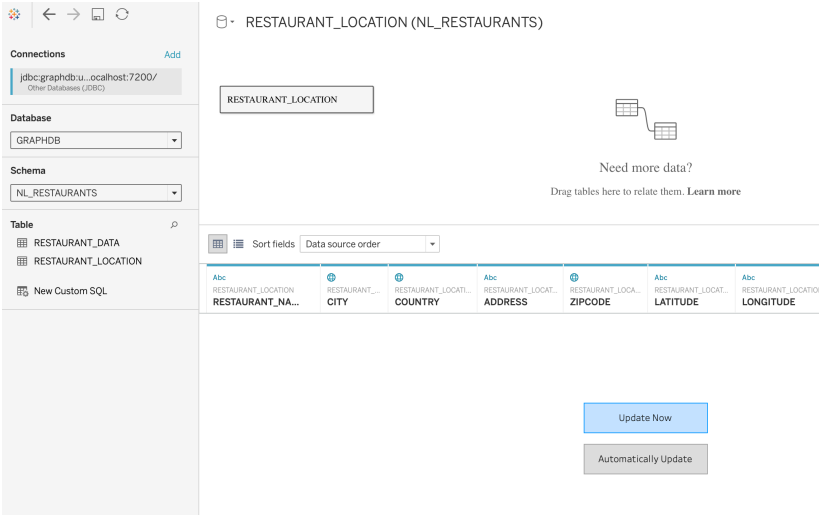
Tableau

Let's transform your RDF data into SQL:

1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-11.3.3.jar).
2. Place it in the in the Tableau directory corresponding to your operating system:
   - Windows: `C:\Program Files\Tableau\Drivers`
   - MacOS: `~/Library/Tableau/Drivers`
3. Start Tableau and go to *Connect* ▸ *Other Databases (JDBC)*.
4. Enter the JDBC connection string in the *URL* field: `jdbc:graphdb:url=http://localhost:7200` (or the respective endpoint URL if your repository is in a remote location).



5. On the next screen, under *Databases* you will see *GraphDB*. Select it.
6. On the drop-down *Schema* menu, you should see the name of the GraphDB repository. If you have configured the SQL view as described above, this will be *NL_Restaurants*. Select it.
7. Tableau is now showing the SQL tables that we created earlier — *restaurant_data* and *restaurant_location*.
8. Drag the *Restaurant_Location* table into the field in the center of the screen and click *Update Now*.

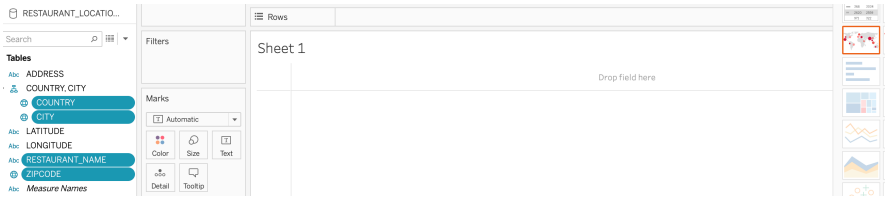


9. Go to **Sheet 1** where we will visualize the restaurants in the dataset based on:

   a. their **location**:

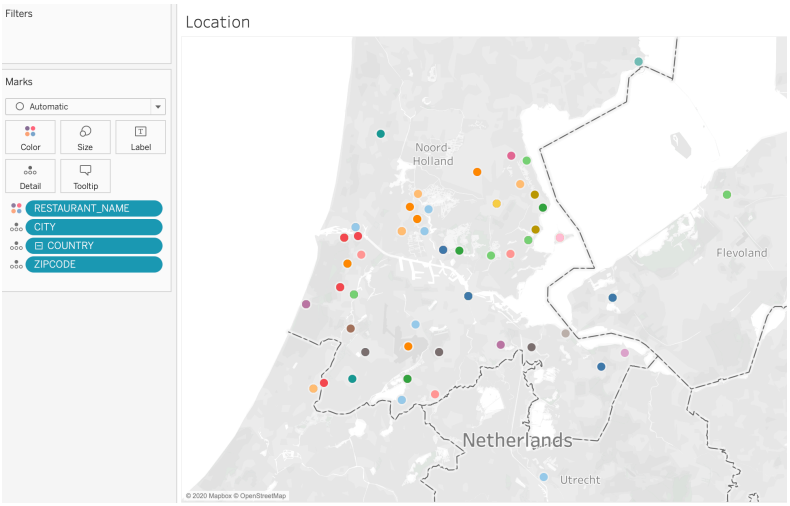
      i. On the left side of the screen, select the parameters: *Country, City, Restaurant_Name, Zipcode*.

      ii. On the right side of the screen, select the *symbol maps* option.



      iii. Drag the *Restaurant_Name* parameter, which is now in the *Rows* field, into *Marks* ▸ *Colors*.

      The resulting map should look like this:



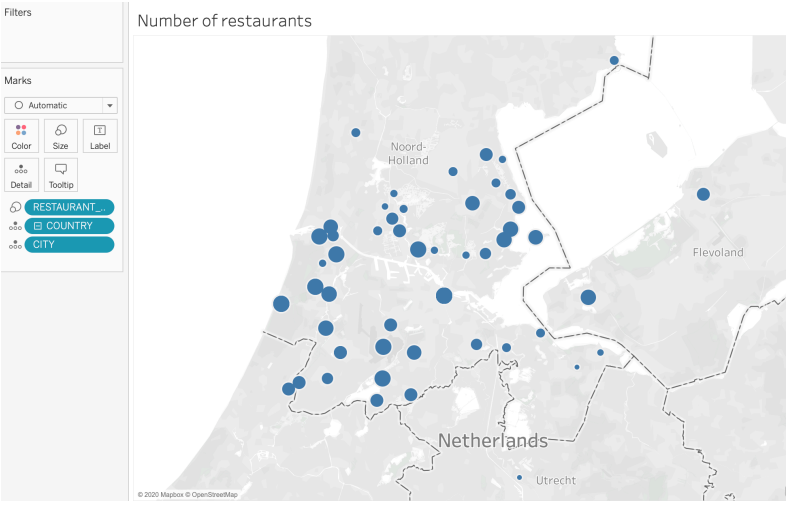
   b. the **number of restaurants in a given location**:

      i. On the left side of the screen, select the parameters: *Country, City, Restaurant_Name*.

      ii. On the right side of the screen, again select the *symbol maps* option.

      iii. Drag the *Restaurant_Name* parameter, which is now in the *Rows* field, into *Marks –> Size*.

The resulting map should look like this:

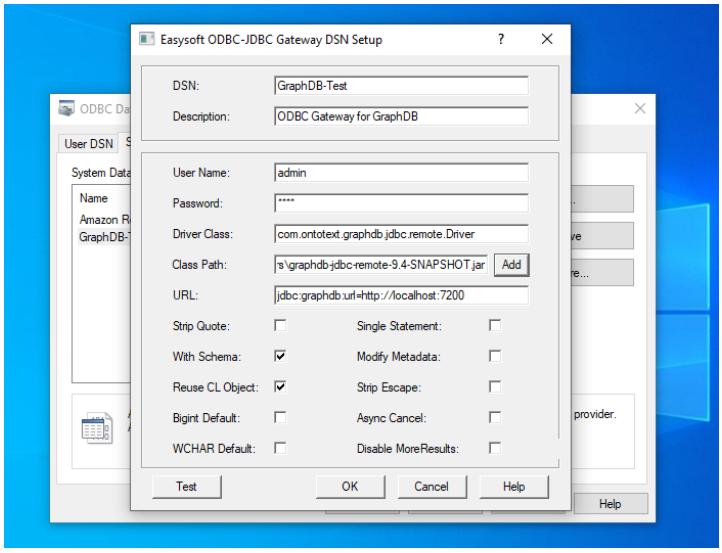


Microsoft Power BI over ODBC protocol

When working with BI tools that do not support JDBC, as is the case with Microsoft Power BI, you need to use an ODBC-JDBC bridge such as Easysoft's ODBC-JDBC Gateway.

After downloading and installing the gateway in your Windows operating system, connect it to GraphDB with these steps:

1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-11.3.3.jar).
2. From the main menu, go to *ODBC Data Sources (64-bit)*.
3. In the dialog box, go to *System DSN* and click *Add*.
4. In the next window, select *Easysoft ODBC-JDBC Gateway* and click *Finish*.
5. In the next window, configure the connection to GraphDB with these values:
    - in the *DSN* field, enter the name of the new driver such as "GraphDB-Test". The *Description* field is optional.
    - for *User Name*, enter "admin", and for *Password* — "root". These are not mandatory, except when GraphDB security is enabled.
    - for *Driver Class*, enter `com.ontotext.graphdb.jdbc.remote.Driver`.
    - for *Class Path*, click *Add* and go to the location of the driver's `.jar` file on your computer. Select it and click *Open*.
    - for *URL*, enter the same string as in the Tableau example above: `jdbc:graphdb:url=http://localhost:7200/` (or the respective endpoint URL if your repository is in a remote location).

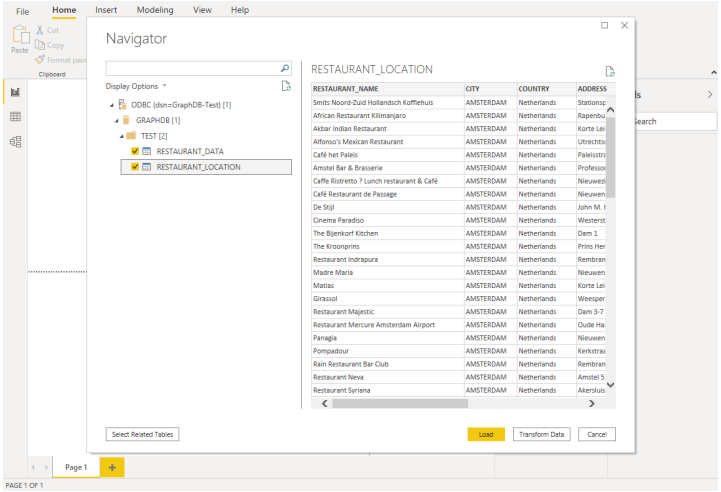


6. Click *Test* to make sure that the connection is working, then click *OK*.
7. In the previous dialog box, you should now see the *GraphDB-Test* connection.

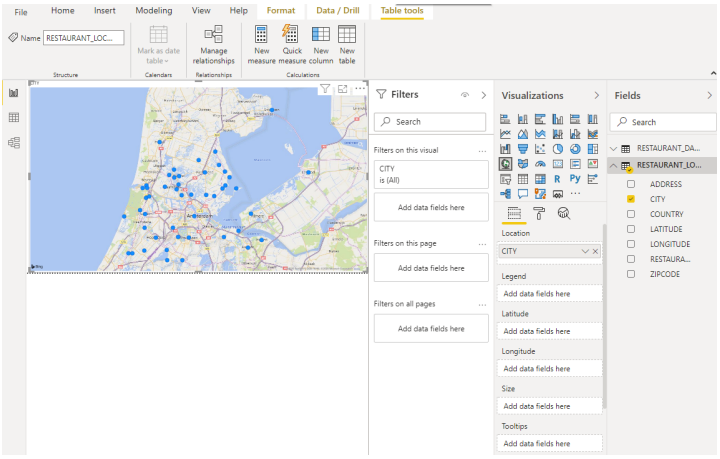
This concludes the gateway configuration, and we are now ready to use it with Microsoft Power BI.

Let's use the Netherlands Restaurants example again:

1. Start Power BI Desktop and go to *Get Data*.
2. From the pop-up *Get Data* window, go to *Other -> ODBC*. Click *Connect*.
3. From drop-down menu in the next dialog, select *GraphDB-Test*.
4. In the next dialog box, enter username "admin" and password "root". (The password is only mandatory if GraphDB security is enabled.)
5. in the Navigator window that appears, you can now see the GraphDB directory and the tables it contains — *Restaurant_Data* and *Restaurant_Location*. Select the tables and click *Load*.



6. To visualize the data as a geographic map (similar to the Tableau example above), select the *Report* option on the left and then the *Map* icon from the *Visualizations* options on the right.
7. You can experiment with the *Fields* that you want visualized. For example: selecting *City* will display all the locations in the dataset.



8. You can also view the data in table format, as well as see the way the two tables are connected, by using the *Data* and *Model* views on the left.

3.4.8.6 How it works: Table description

As mentioned above, each SQL table is described by a SPARQL query that also includes some metadata defining the SQL columns, their types, and the expected RDF type. For the *restaurant_data* example, it will look like this:

```
# !column : restaurant_name : string not null
# !column : short_description  : string
# !column : long_description  : string
# !column : calendar  : string

PREFIX ex:<http://example.com/ex>
PREFIX base:<http://example/base/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

select ?restaurant_name ?short_description ?long_description ?calendar where {
    ?s a base:Restaurant;
        rdfs:label ?restaurant_name;
    ex:shortDescription ?short_description;
    ex:longDescription ?long_description;
    ex:calendar ?calendar.
    # !filter
}
```

It is generated as an `.rq` file upon creation of a SQL table from Workbench, and is automatically saved in a newly created `sql` subdirectory in the respective repository folder. In our case, this would be:

`data/repositories/nl_restaurants/sql/restaurant_data`

You can download and have a look at the two SPARQL queries that we used for the above examples:

- `restaurant_data.rq`
- `restaurant_location.rq`

3.4.9 Query profiling with the Explain plan

GraphDB's Explain plan is a feature that explains how GraphDB executes a SPARQL query. It also includes information about unique subject, predicate and object collection sizes. This can help you improve your query, leading to better execution performance.

3.4.9.1 Activating the Explain plan

To see the query Explain plan, use the `onto:explain` pseudo-graph:

```
PREFIX onto: <http://www.ontotext.com/>
SELECT * FROM onto:explain
```

Note: Using `FROM onto:measure` instead of `FROM onto:explain` will execute the query and also collect the number of steps and timings for each step, showing the precise execution process. It provides these for the evaluation groups and the patterns included. This more precise data enables further investigation about potential bottlenecks during query processing.

3.4.9.2 A simple Explain plan

For the simplest query Explain plan possible (`?s ?p ?o`), execute the following query:

```
PREFIX onto: <http://www.ontotext.com/>
SELECT * FROM onto:explain {
   ?s ?p ?o .
}
```

Depending on the number of triples that you have in the database, the results will vary, but you will get something like the following:

plan

```
1 SELECT ?s ?p ?o
  {

    { # ----- Begin optimization group 1 -----

      ?s ?p ?o . #    Collection size: 70 Predicate collection size: 70   Unique subjects: 60 Unique objects: 36  Current complexity: 70

    } # ----- End optimization group 1 -----
    # ESTIMATED NUMBER OF ITERATIONS: 70

  }

  # NOTE: Optimization groups are evaluated one after another exactly in the given order.
  # If there are too many optimization groups consisting of a single statement pattern,
  # then one should try to relocate the following clauses by hand:
  # VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
  # Sub-SELECTs will always be evaluated first.
```

This is the same query, but with some estimations next to the statement pattern (1 in this case).

Note: The query might not be the same as the original one. Below you can see the triple patterns in the order in which they are executed internally.

- `----- Begin optimization group 1 -----`: indicates the start of a group of statements, which most probably are part of a subquery (in the case of property paths, the group will be the whole path).
- `Collection size`: an estimation of the number of statements that match the pattern.
- `Predicate collection size`: the number of statements in the database for this particular predicate (in this case, for all predicates).

- `Unique subjects`: the number of subjects that match the statement pattern.
- `Unique objects`: the number of objects that match the statement pattern.
- `Current complexity`: the complexity (the number of atomic lookups in the index) the database will need to make so far in the optimization group — most of the time, a subquery. When you have multiple triple patterns, these numbers grow fast.
- `----- End optimization group 1 -----`: the end of the optimization group.
- `ESTIMATED NUMBER OF ITERATIONS`: the approximate number of iterations that will be executed for this group. For `onto:measure`, this is the actual number of iterations.

3.4.9.3 Multiple triple patterns

Note: The result of the Explain plan is given in the exact order in which the engine will execute the query.

In the following example, the engine reorders the triple patterns based on their complexity. The query is a simple join:

```
PREFIX onto: <http://www.ontotext.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

select *
from onto:explain
{
    ?o rdf:type ?o1 .
    ?o rdfs:subPropertyOf ?o2
}
```

The output of this example query is shown below:

```
plan
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

  SELECT ?o ?o1 ?o2
  {

    { # ----- Begin optimization group 1 -----

      ?o rdfs:subPropertyOf ?o2 . #   Collection size: 10 Predicate collection size: 10   Unique subjects: 10 Unique objects: 8   Current complexity: 10
      ?o rdf:type ?o1 . #     Collection size: 30 Predicate collection size: 30   Unique subjects: 21 Unique objects: 7   Current complexity: 14.285714

    } # ----- End optimization group 1 -----
    # ESTIMATED NUMBER OF ITERATIONS: 14.285714

  }

  # NOTE: Optimization groups are evaluated one after another exactly in the given order.
  # If there are too many optimization groups consisting of a single statement pattern,
  # then one should try to relocate the following clauses by hand:
  # VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
  # Sub-SELECTs will always be evaluated first.
```

Some highlights of the output:

- `?o rdfs:subPropertyOf ?o2` has a lower collection size (10 instead of 30), so it will be executed first.
- `?o rdf:type ?o1` has a bigger collection size (30 instead of 10), so it will be executed second, although it is written first in the original query.
- The current complexity grows fast because it multiplies. In this case, you can expect to get 10 results from the first statement pattern. Then you need to join them with the results from the second triple pattern, which results in the complexity of `10 * 30 = 300`.
- Although the complexity for the whole group is `300`, the estimated number of iterations for this group is `14.3`.

3.4.9.4 Sample wine queries

All of the following examples are based on this simple **dataset** describing five fictitious wines. The file is quite small and tells us the following about these wines:

- There are different types of wine (Red, White, Rose).
- Each wine has a label.
- Wines are made from different types of grapes.
- Wines contain different levels of sugar.
- Wines are produced in a specific year.

Querying with aggregation

A typical aggregation query contains a group with some aggregation function. Here, we have added an **explain** graph.

This query retrieves the number of wines produced in each year along with the year.

```
PREFIX onto: <http://www.ontotext.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wine: <http://www.example.com/demo/wine#>
SELECT (COUNT(?wine) as ?wines) ?year
FROM onto:explain
WHERE {
    ?wine rdf:type wine:Wine .
    OPTIONAL {
        ?wine wine:hasYear ?year
    }
}
GROUP BY ?year
ORDER BY DESC(?wines)
```

When you execute the query in GraphDB, you get the following as output instead of the actual results:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT (COUNT(?wine) AS ?wines) ?year
{

  { # ----- Begin optimization group 1 -----

    ?wine rdf:type wine:Wine . #    Collection size: 5  Predicate collection size: 50   Unique subjects: 36 Unique objects: 12  Current complexity: 5

  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 5

  OPTIONAL
  {

    { # ----- Begin optimization group 2 -----

      ?wine wine:hasYear ?year . #  Collection size: 5  Predicate collection size: 5    Unique subjects: 5  Unique objects: 2   Current complexity: 1

    } # ----- End optimization group 2 -----
    # ESTIMATED NUMBER OF ITERATIONS: 1

  }
}
GROUP BY ?year
ORDER BY DESC(?wines)

# NOTE: Optimization groups are evaluated one after another exactly in the given order.
# If there are too many optimization groups consisting of a single statement pattern,
# then one should try to relocate the following clauses by hand:
# VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
# Sub-SELECTs will always be evaluated first.
```

Querying with filter aggregation

This aggregation query groups the results and then applies filtering with a `HAVING` clause. It retrieves red wines made from more than one type of grape along with their grapes count.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX onto: <http://www.ontotext.com/>
SELECT ?wine (COUNT(?grape) AS ?grapeCount)
FROM onto:explain
WHERE {
    ?wine rdf:type wine:RedWine ;
          wine:madeFromGrape ?grape .
}
GROUP BY ?wine
HAVING (?grapeCount > 1)
```

The returned Explain plan will be:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?wine (COUNT(?grape) AS ?grapeCount)
{

  { # ----- Begin optimization group 1 -----

    ?wine rdf:type wine:RedWine . #     Collection size: 3  Predicate collection size: 50   Unique subjects: 36 Unique objects: 12  Current complexity: 3
    ?wine wine:madeFromGrape ?grape . #     Collection size: 6  Predicate collection size: 6    Unique subjects: 5  Unique objects: 5   Current complexity: 3.6

  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 3.6

  FILTER (?grapeCount > 1)
  BIND (COUNT(?grape) AS ?grapeCount)
}
GROUP BY ?wine

# NOTE: Optimization groups are evaluated one after another exactly in the given order.
# If there are too many optimization groups consisting of a single statement pattern,
# then one should try to relocate the following clauses by hand:
# VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
# Sub-SELECTs will always be evaluated first.
```

Querying with a filter function

This is a typical SPARQL query with a filter function. It retrieves the wines that are made from the Pinot Noir grape.

```
PREFIX onto: <http://www.ontotext.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?wine ?sugar ?year ?grapeLabel
FROM onto:explain
WHERE {
    ?wine rdf:type wine:Wine ;
          wine:hasSugar ?sugar ;
          wine:hasYear ?year ;
          wine:madeFromGrape ?grape .
    ?grape rdfs:label ?grapeLabel .
    FILTER (?grapeLabel = "Pinot Noir")
}
```

The output shows how the optimization converted the filter expression to a simple triple pattern in the query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?wine ?sugar ?year ?grapeLabel
{

  { # ----- Begin optimization group 1 -----

    ?grape rdfs:label "Pinot Noir" . #  Collection size: 1  Predicate collection size: 8    Unique subjects: 8  Unique objects: 8   Current complexity: 1
    ?wine wine:madeFromGrape ?grape . #     Collection size: 6  Predicate collection size: 6    Unique subjects: 5  Unique objects: 5   Current complexity: 1.2
    ?wine wine:hasSugar ?sugar . #  Collection size: 5  Predicate collection size: 5    Unique subjects: 5  Unique objects: 2   Current complexity: 1.2
    ?wine wine:hasYear ?year . #    Collection size: 5  Predicate collection size: 5    Unique subjects: 5  Unique objects: 2   Current complexity: 1.2
    ?wine rdf:type wine:Wine . #    Collection size: 5  Predicate collection size: 50   Unique subjects: 36 Unique objects: 12  Current complexity: 1.2

  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 1.2

}

# NOTE: Optimization groups are evaluated one after another exactly in the given order.
# If there are too many optimization groups consisting of a single statement pattern,
# then one should try to relocate the following clauses by hand:
# VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
# Sub-SELECTs will always be evaluated first.
```

3.4.10 Change tracking

GraphDB lets you track changes that you have made in your data. Two tools offer this capability: the change tracking plugin and the *data history and versioning plugin*.

3.4.10.1 What the plugin does

The change tracking plugin is useful for tracking changes within the context of a transaction identified by a unique ID. Different IDs allow tracking of multiple independent changes — for example, user A tracks his updates and user B tracks her updates without interfering with each other. The tracked data is stored only in memory and is not available after a restart.

As part of the GraphDB Plugin API, the change tracking plugin lets you track the effect of SPARQL updates. These can be:

- Tracking what triples have been inserted or deleted
- Distinguishing explicit from implicit triples
- Running SPARQL using these triples

3.4.10.2 Usage

The plugin uses the following special graphs:

- `http://www.ontotext.com/added/xxx` — all added statements, including inferred ones
- `http://www.ontotext.com/removed/xxx` — all removed statements, including inferred ones

In both cases, `xxx` is a user-provided unique ID that must be assigned when activating the tracking function.

The usage pattern goes like this:

1. Start a transaction.
2. Enable tracking for this transaction, where `xxx` is a unique ID assigned by the user:

```
INSERT DATA {
   [] <http://www.ontotext.com/track-changes> "xxx"
}
```

3. Add some data (or remove some with the `DELETE DATA` equivalent of the update request below):

```
INSERT DATA {
   [] <http://www.ontotext.com/track-changes> "xxx"
};
INSERT DATA {<urn:a> <urn:b> <urn:c>};
```

Note: All queries must be executed in the same SPARQL editor.

4. Commit the transaction.
5. Retrieve all added triples and their graph:

```
SELECT * FROM <http://www.ontotext.com/added/xxx> {
   graph ?g {
       ?s ?p ?o
   }
}
```

6. Retrieve the number of removed triples:

```
SELECT (COUNT(*) as ?c) FROM <http://www.ontotext.com/removed/xxx> {
   ?s ?p ?o
}
```

7. Execute a `CONSTRUCT` query using data that has just been added (advanced example):

```
BASE <http://ontotext.com/resource/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX test: <http://ontotext.com/vocabulary/test/>

CONSTRUCT {
   ?person test:knows ?knows ;
           foaf:givenName ?givenName
} FROM <http://www.ontotext.com/added/xxx> WHERE {
   ?person foaf:givenName ?givenName ;
           foaf:knows     ?knows
}
```

8. Forget the tracked data:

```
INSERT DATA {
   <http://www.ontotext.com/track-changes> <http://www.ontotext.com/delete-changes> "xxx
↪"
}
```

Note: You must explicitly delete the tracked changes when you no longer need to query them. Otherwise they will stay in memory until the same ID is used again, or until GraphDB is restarted.

Tip: A good way to ensure unique tracking IDs is to use UUIDs. A random UUID can be generated in Java by calling `UUID.randomUUID().toString()`.

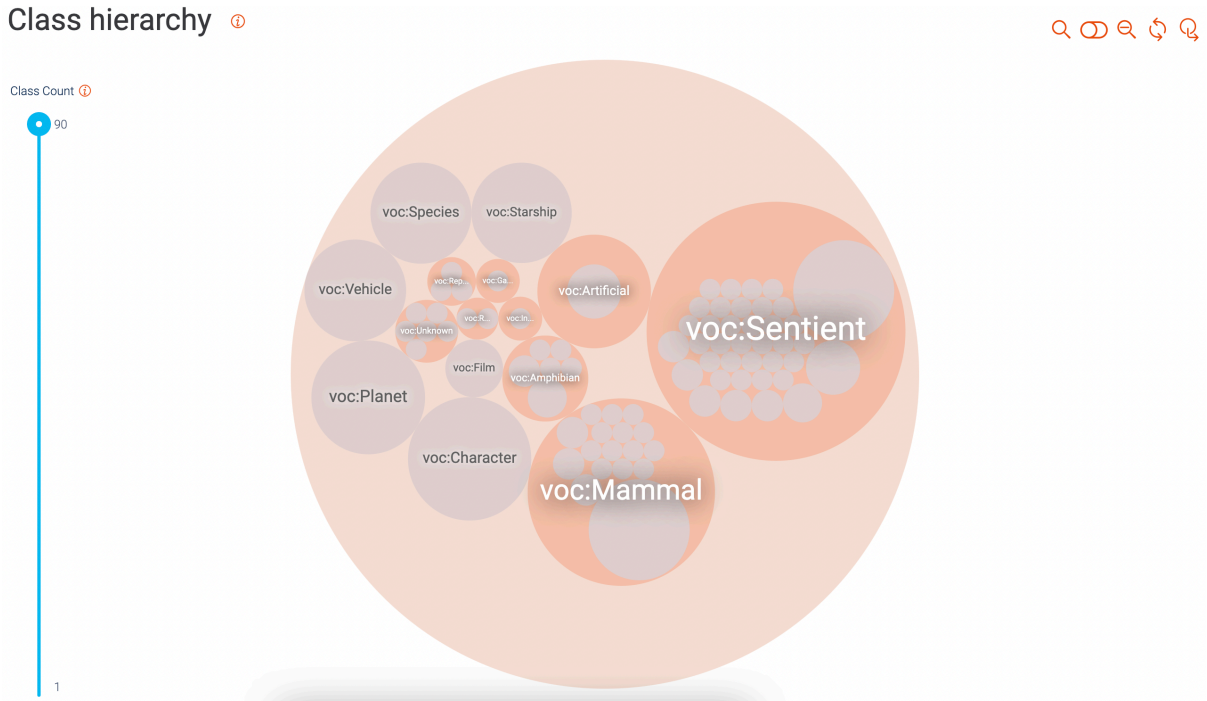
3.5 Visualize and explore

This guide uses a variation of the `Star Wars dataset` that you can download and use to execute the examples yourself.

3.5.1 Class hierarchy

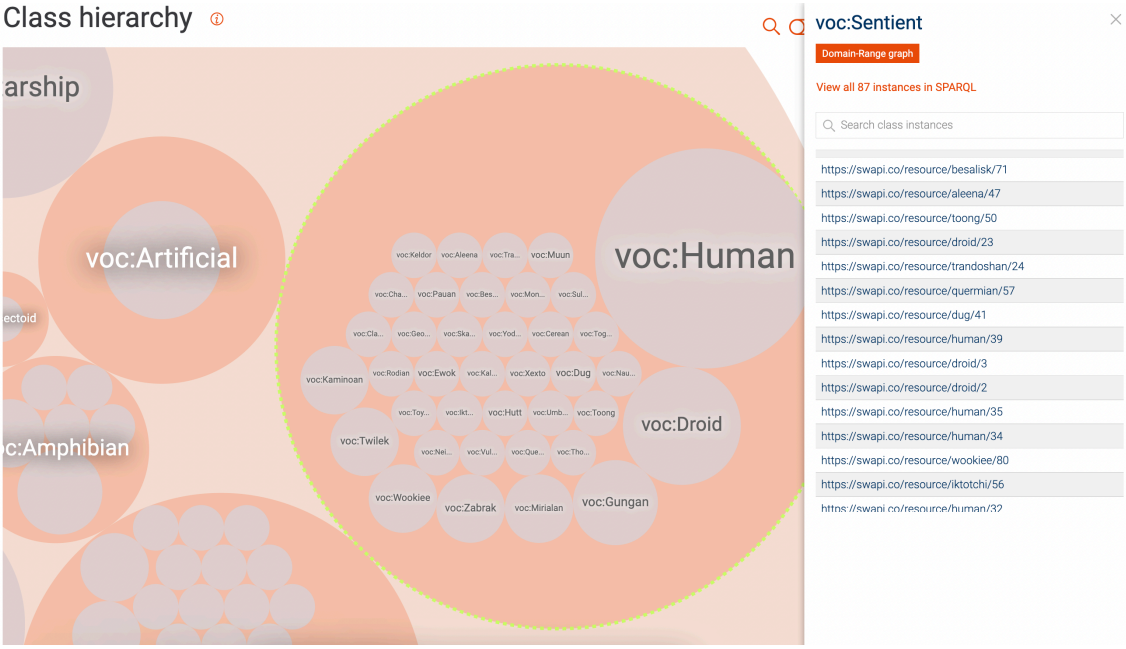
To explore your data, navigate to *Explore* ▸ *Class hierarchy*. You can see a diagram depicting the hierarchy of the imported RDF classes by the number of instances. The biggest circles are the parent classes and the nested ones are their children.

Note: If your data model is not hierarchical, the RDF classes are visualized as separate circles instead of nested ones.



3.5.1.1 Explore your data — different actions

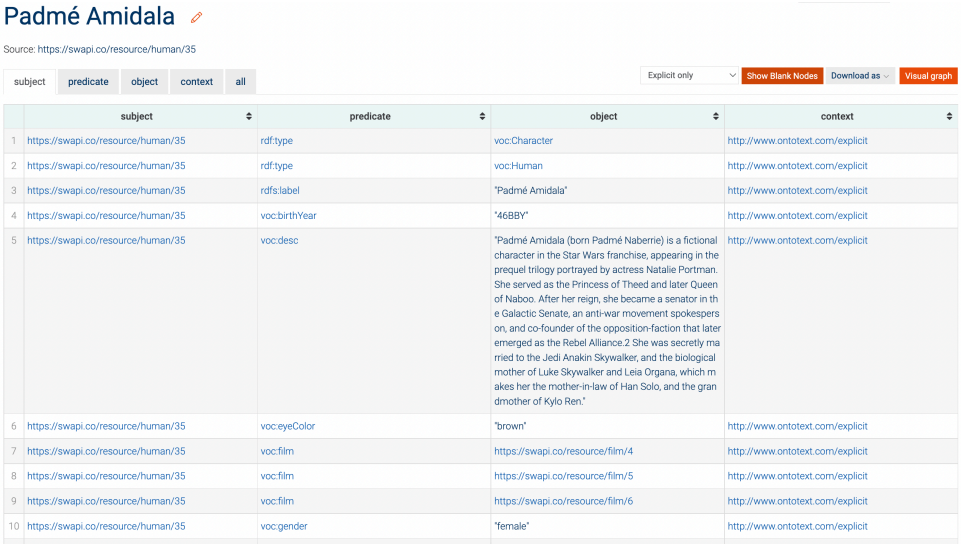
- To see what classes each parent has, hover over the nested circles.
- To explore a given class, click its circle. The selected class is highlighted with a dashed line and a side panel with its instances opens for further exploration. For each RDF class, you can see its local name, IRI, and a list of its first 1,000 class instances. The class instances are represented by their IRIs, which, when clicked, lead to another view where you can further explore their metadata.



Note: The side panel includes the following:

- Local name
- IRI (Press `Ctrl+C` / `Cmd+C` to copy to clipboard and `Enter` to close)
- *Domain-Range Graph* button
- Class instances count
- Scrollable list of the first 1,000 class instances
- A *View Instances in SPARQL View* button; this redirects to the SPARQL view and executes an auto-generated query that lists all class instances without `LIMIT`

- To go to the *Domain-Range Graph* diagram, double-click a class circle or the *Domain-Range Graph* button from the side panel.
- To explore an instance, click its IRI from the side panel. The following shows an example of an instance display:



Padmé Amidala

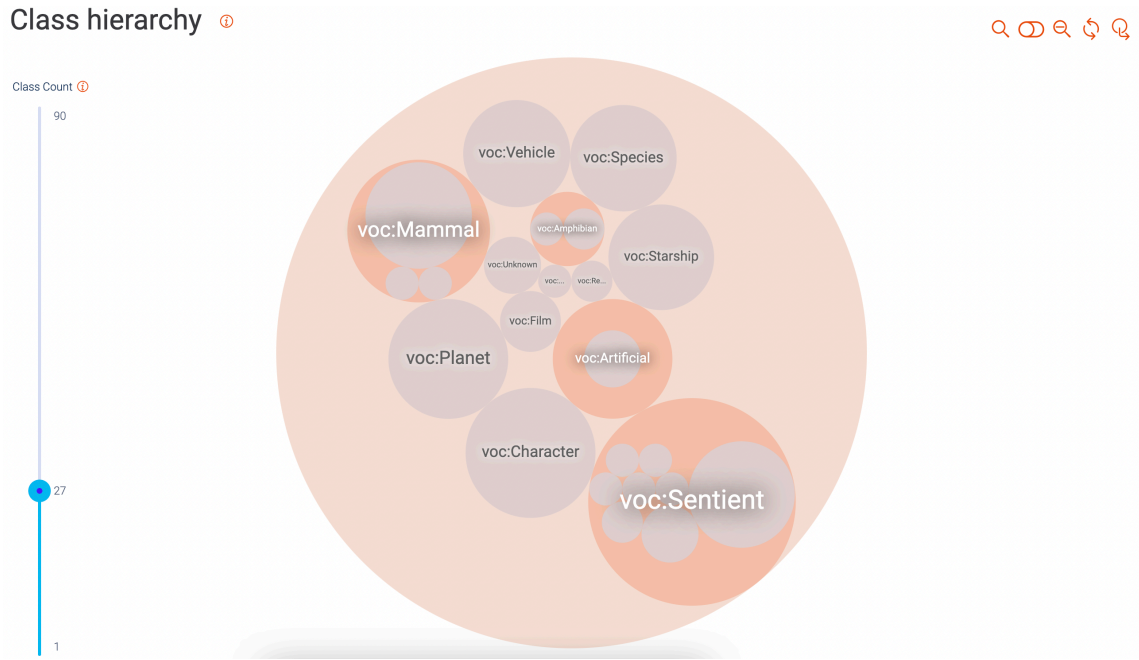
Source: https://swapi.co/resource/human/35

subject | predicate | object | context | all

Explicit only | Show Blank Nodes | Download as | Visual graph

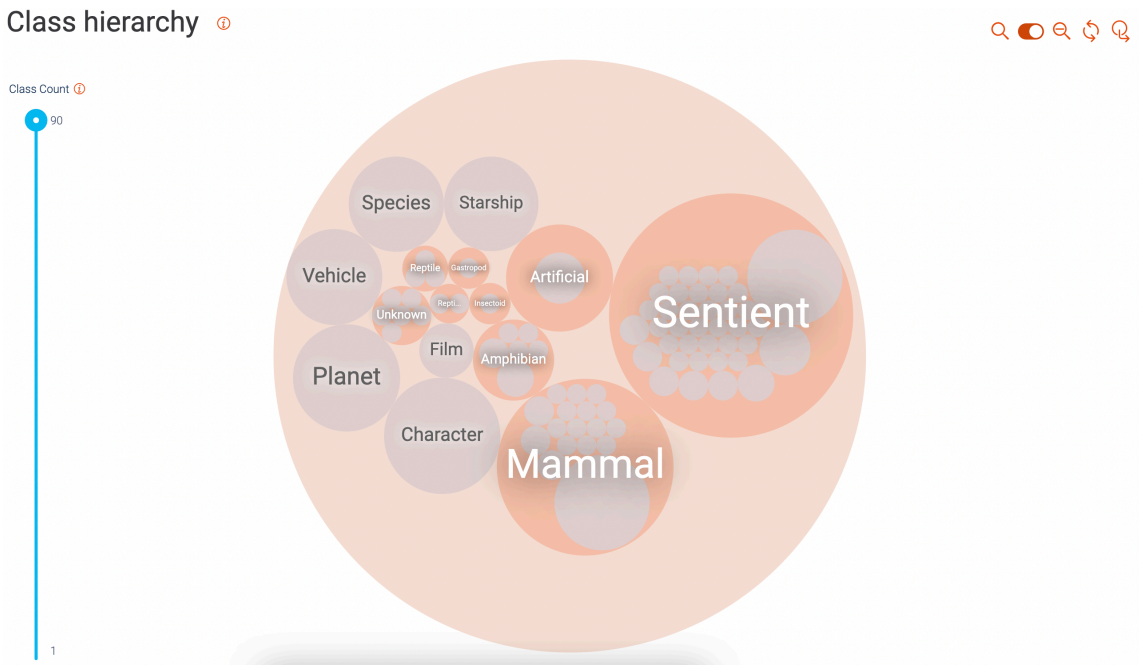
	subject	predicate	object	context
1	https://swapi.co/resource/human/35	rdf:type	voc:Character	http://www.ontotext.com/explicit
2	https://swapi.co/resource/human/35	rdf:type	voc:Human	http://www.ontotext.com/explicit
3	https://swapi.co/resource/human/35	rdfs:label	"Padmé Amidala"	http://www.ontotext.com/explicit
4	https://swapi.co/resource/human/35	voc:birthYear	"46BBY"	http://www.ontotext.com/explicit
5	https://swapi.co/resource/human/35	voc:desc	"Padmé Amidala (born Padmé Naberrie) is a fictional character in the Star Wars franchise, appearing in the prequel trilogy portrayed by actress Natalie Portman. She served as the Princess of Theed and later Queen of Naboo. After her reign, she became a senator in th e Galactic Senate, an anti-war movement spokespers on, and co-founder of the opposition-faction that later emerged as the Rebel Alliance.2 She was secretly ma rried to the Jedi Anakin Skywalker, and the biological mother of Luke Skywalker and Leia Organa, which m akes her the mother-in-law of Han Solo, and the gran dmother of Kylo Ren."	http://www.ontotext.com/explicit
6	https://swapi.co/resource/human/35	voc:eyeColor	"brown"	http://www.ontotext.com/explicit
7	https://swapi.co/resource/human/35	voc:film	https://swapi.co/resource/film/4	http://www.ontotext.com/explicit
8	https://swapi.co/resource/human/35	voc:film	https://swapi.co/resource/film/5	http://www.ontotext.com/explicit
9	https://swapi.co/resource/human/35	voc:film	https://swapi.co/resource/film/6	http://www.ontotext.com/explicit
10	https://swapi.co/resource/human/35	voc:gender	"female"	http://www.ontotext.com/explicit

- To adjust the number of classes displayed, drag the slider on the left side of the screen. Classes are sorted by the maximum instance count, and the diagram displays only the *current slider value*.



- To administrate your data view, use the toolbar options on the right side of the screen.

  - To see only the class labels, click the *Hide/Show Prefixes*. You can still view the prefixes when you hover over the class that interests you.

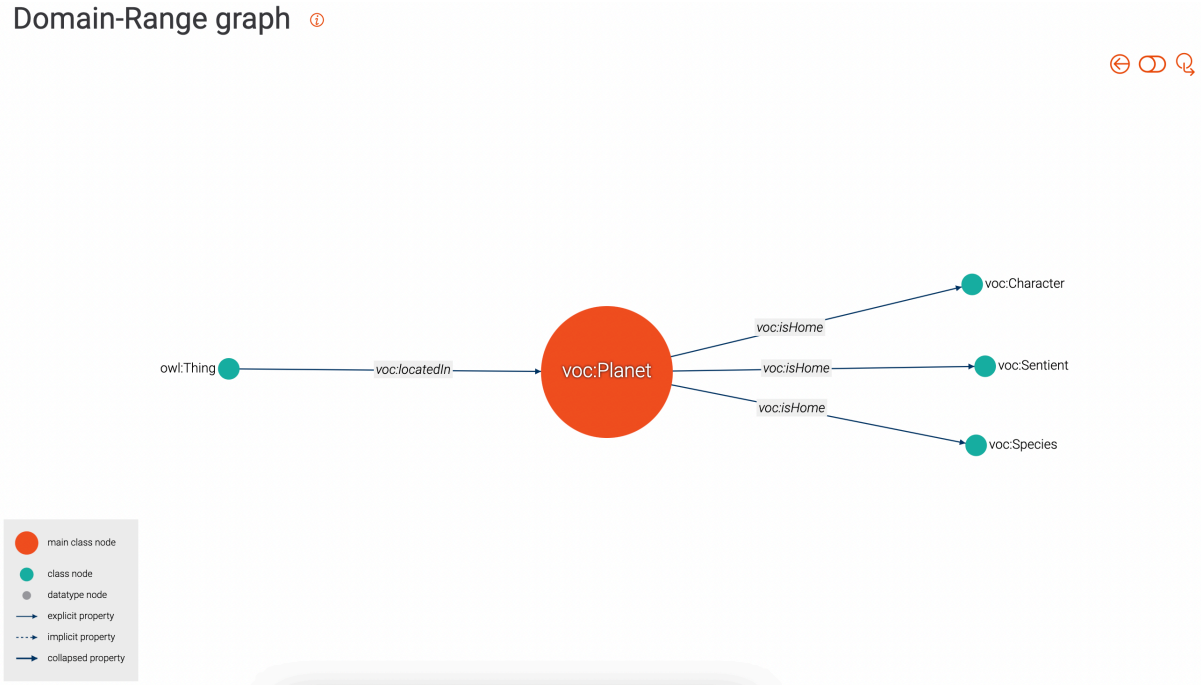


- To zoom out of a particular class, click the *Focus diagram* icon.
- To reload the data used to generate the diagram, click the *Reload diagram* icon. This is recommended when you have updated the data in your repository or when you are experiencing strange behavior such as a class being missing from the display.
- To export the diagram as an `.svg` image, click the *Export Diagram* download icon.

- You can also filter the hierarchy by graph when there is more than one named graph in your repository. To do this, expand the *All graphs* drop-down menu next to the toolbar options and select the graph you want to explore.

3.5.1.2 Domain-range graph

To see all properties of a given class as well as their domain and range, double-click its class circle or the *Domain-Range Graph* button from the side panel. The RDF Domain-Range Graph view opens, letting you further explore the class connectedness by clicking the green object property class nodes.



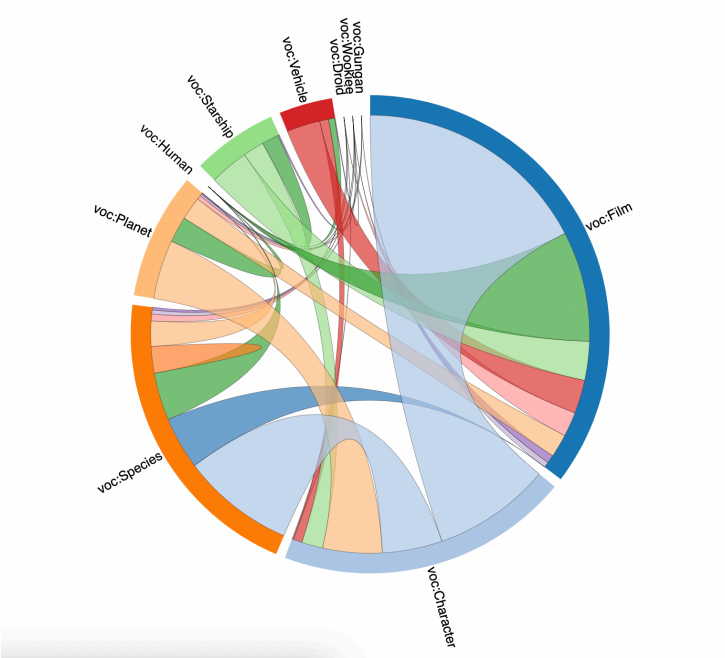
- To administrate your graph view, use the toolbar options on the right side of the screen.

  - To go back to your class in the RDF Class hierarchy, click the *Back to Class hierarchy* diagram button.
  - To export the diagram as an `.svg` image, click the *Export Diagram* download icon.

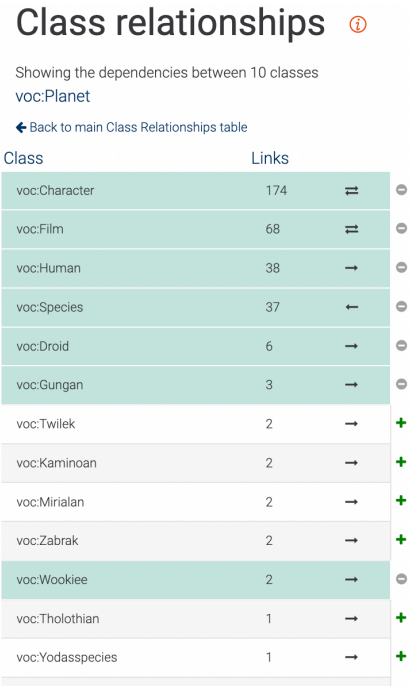
3.5.2 Class relationships

To explore the relationships between the classes, navigate to *Explore* ‣ *Class relationships*. This displays a complicated diagram, which by default shows only the top relationships. Each of them is a bundle of links between the individual instances of two classes. Each link is an RDF statement where the subject is an instance of one class, the object is an instance of another class, and the link is the predicate. Depending on the number of links between the instances of two classes, the bundle can be thicker or thinner and has the color of the class with more incoming links. These links can go in both directions. Note that contrary to the *Class hierarchy*, the *Class relationships* diagram is based on the real statements between classes and not on the ontology schema.

In the example below, we can see that “Character” is the class with the biggest number of links. It is very strongly connected to “Film” and “Species”, and most of the links are to “Character”.



To the left of this diagram, you can see a list of all classes ordered by the number of links they have, as well as an indicator of the direction of the links. Click it to see a list of the classes that this class links to, sorted by the number of links. The display includes the direction of the links.



Class relationships

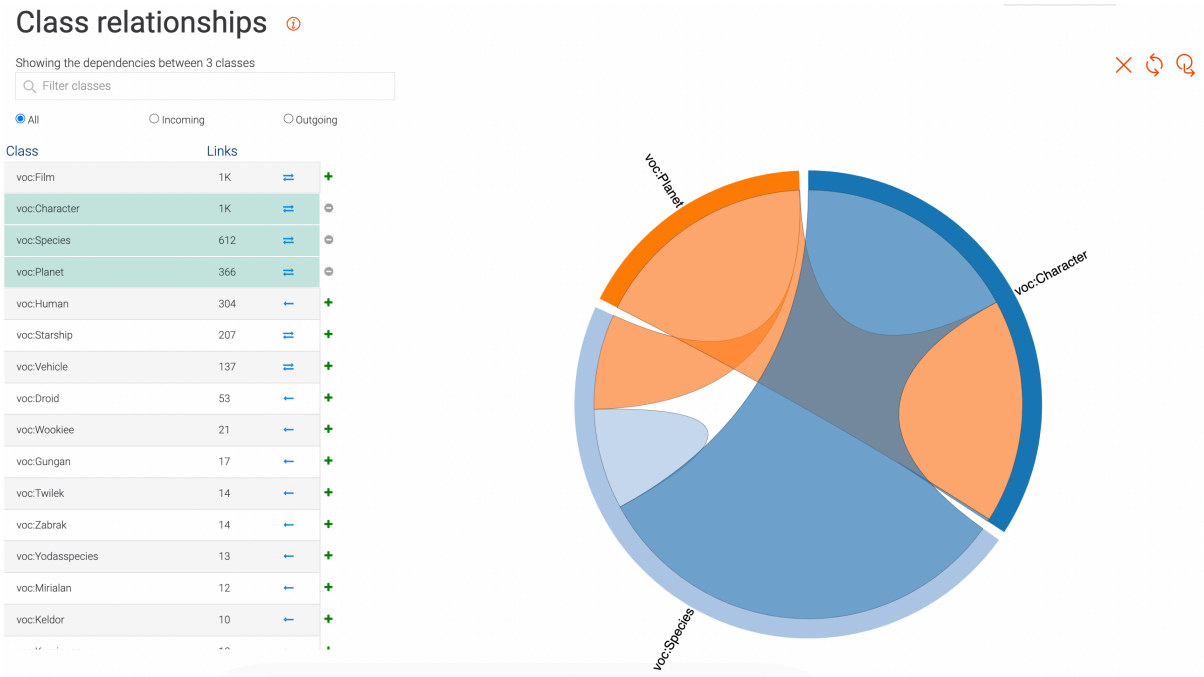
Showing the dependencies between 10 classes

voc:Planet

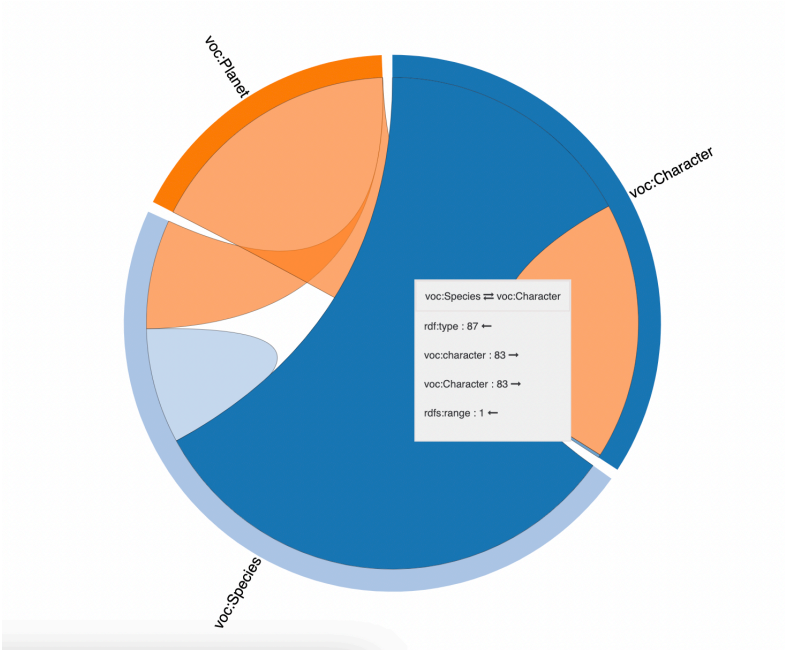
Back to main Class Relationships table

Class	Links		
voc:Character	174	⇄	−
voc:Film	68	⇄	−
voc:Human	38	→	−
voc:Species	37	←	−
voc:Droid	6	→	−
voc:Gungan	3	→	−
voc:Twilek	2	→	+
voc:Kaminoan	2	→	+
voc:Mirialan	2	→	+
voc:Zabrak	2	→	+
voc:Wookiee	2	→	−
voc:Tholothian	1	→	+
voc:Yodasspecies	1	→	+

Use the add and remove icons to the right of each listed class name to control which classes are displayed in the diagram. You can remove all the classes with the **X** icon on the top right of the diagram. A green background behind a class name indicates that the class is present in the diagram. Above, we see that “Planet” has many more connections to “Character” than to “Species”.



For every pair of classes in the diagram, you can find the top predicates that connect them by clicking on the connection. The display will sort these by the number of statements using this predicate and instances of the classes.



Just like in the *Class hierarchy* view, you can also filter the class relationships by graph when there is more than one named graph in the repository. Expand the *All graphs* drop-down menu next to the toolbar options and select the graph you want to explore.

Note: All of these statistics are built on top of the whole repository, so when you have a lot of data, the building of the diagram may be fairly slow.

You can also use SPARQL queries to explore the class relationships of your data. For example, execute the following query from GraphDB's *SPARQL* view:

```
PREFIX deps: <http://www.ontotext.com/plugins/dependencies#>

select ?typeSubj ?predicate ?typeObj ?count {
    _:b deps:listPredicates '' ;
        deps:fromClass ?typeSubj ;
        deps:toClass ?typeObj ;
        deps:predicate ?predicate ;
        deps:predicateCount ?count .

} order by DESC(?count) ?typeSubj ?predicate ?typeObj
```

This will return:

Filter query results

Showing results from 1 to 224 of 224. Query took 0.1s, minutes ago.

	typeSubj	predicate	typeObj	count
1	voc:Character	voc:film	voc:Film	"173"^^xsd:long
2	voc:Film	voc:Character	voc:Character	"173"^^xsd:long
3	voc:Film	voc:character	voc:Character	"173"^^xsd:long
4	voc:Character	rdf:type	voc:Species	"87"^^xsd:long
5	voc:Character	voc:homeworld	voc:Planet	"87"^^xsd:long
6	voc:Planet	voc:resident	voc:Character	"87"^^xsd:long
7	voc:Species	voc:Character	voc:Character	"83"^^xsd:long
8	voc:Species	voc:character	voc:Character	"83"^^xsd:long
9	voc:Film	voc:Character	voc:Human	"81"^^xsd:long
10	voc:Film	voc:character	voc:Human	"81"^^xsd:long
11	voc:Species	voc:film	voc:Film	"77"^^xsd:long

3.5.3 Explore resources

3.5.3.1 Explore resources through the easy graph

Note: Before exploring resources from this view, make sure you have enabled the Autocomplete index for this repository from *Setup* ▸ *Autocomplete*.

Navigating to *Explore* ▸ *Visual graph*. *Easy graph* enables you to explore the graph of your data without using SPARQL. This displays a search input field where you can enter, using autocomplete, the IRI of a resource to use as a starting point for graph exploration.

Note: Any visual graph that a particular user creates will be available with all repositories that that user has access to.

Visual graph

Easy graph

Search for an IRI to view it and configure the visualisation through the UI without using SPARQL

droid

Show

Hint: "abC" matches "abC*", "ab c*" and "ab-c*"

Droid <https://swapi.co/vocabulary/Droid>

Droid gunship <https://swapi.co/resource/vehicle/73>

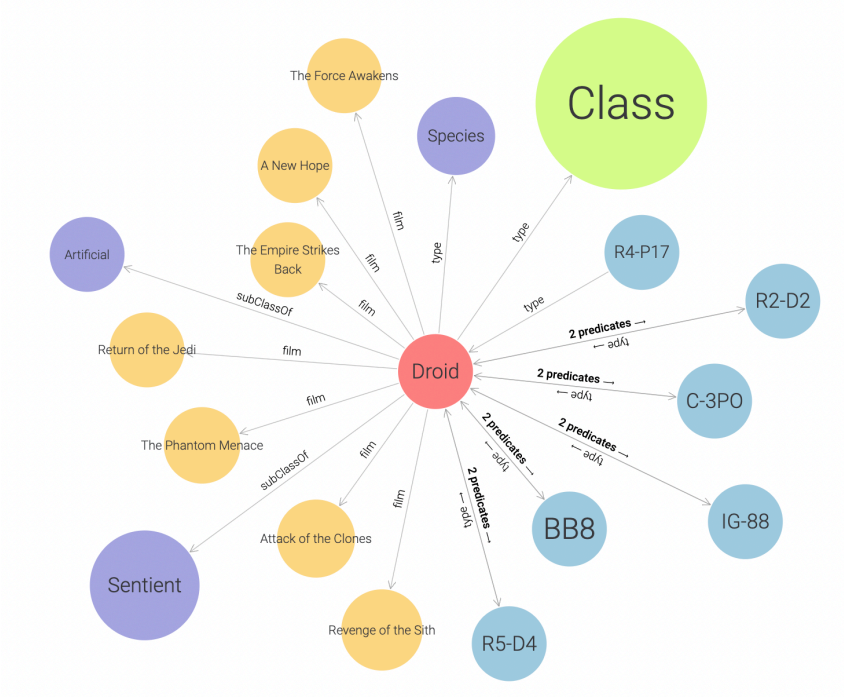
Droid tri-fighter <https://swapi.co/resource/vehicle/67>

Droid control ship <https://swapi.co/resource/starship/32>

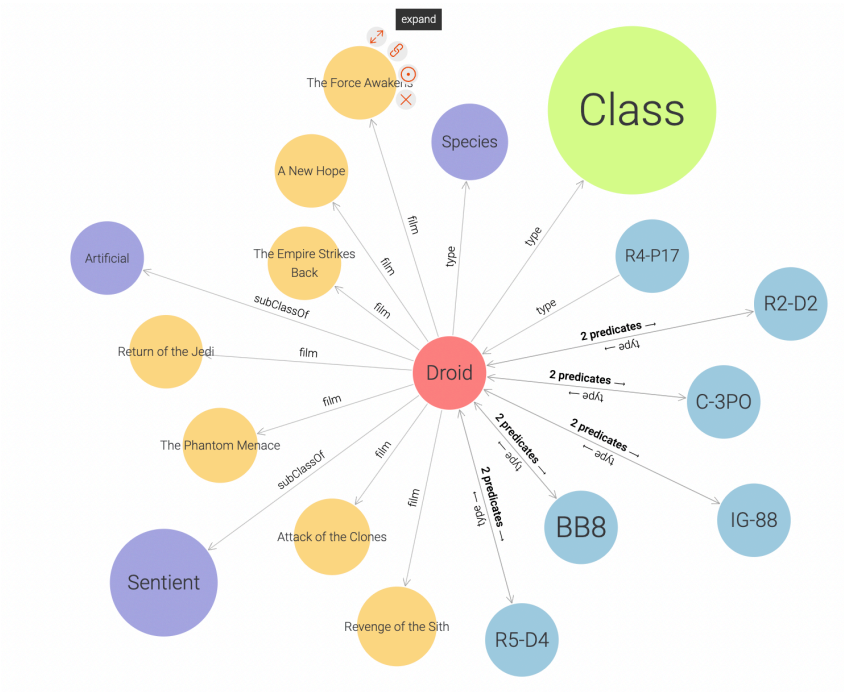
Vulture Droid <https://swapi.co/resource/vehicle/33>

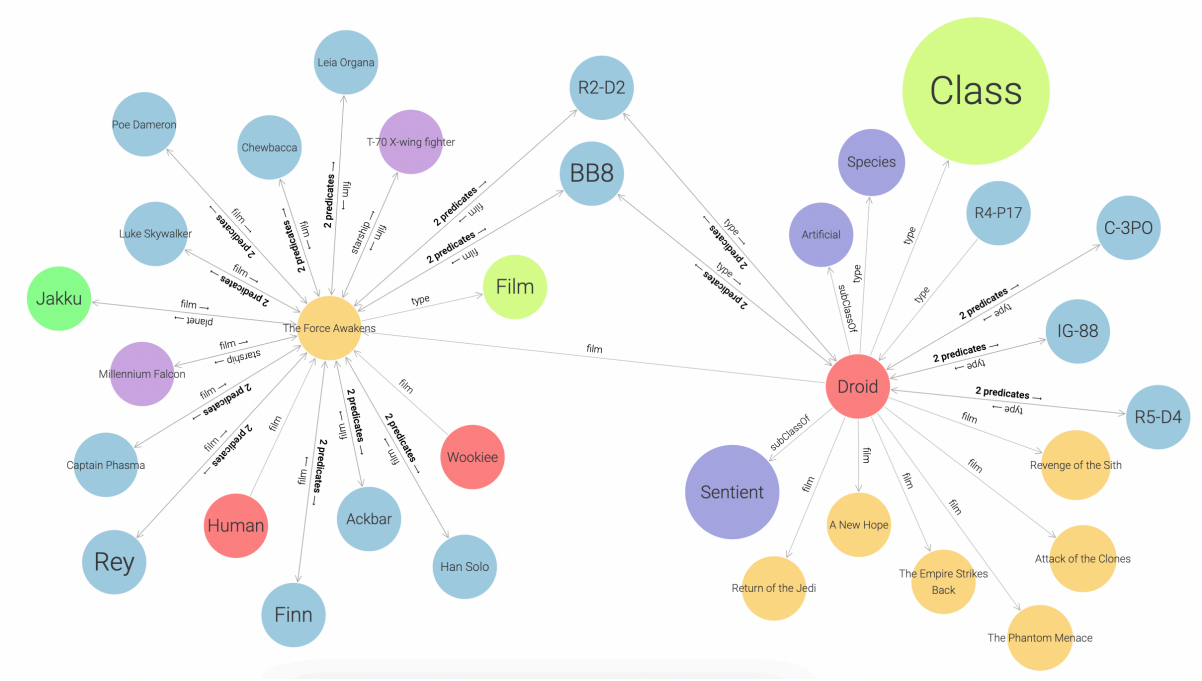
Corporate Alliance tank droid <https://swapi.co/resource/vehicle/72>

This displays a graph of the resource links, with all nodes of a given type having the same color. Hovering over a node displays all of its types. By default, the diagram displays the first 20 links to other resources, sorted by RDF rank if present. See the settings below to modify this limit and the types and predicates to hide or display.

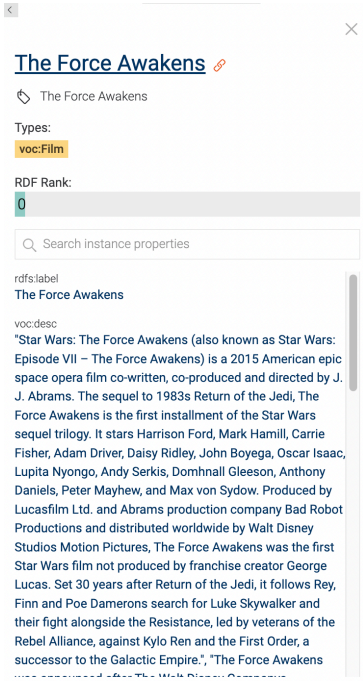


A node's size indicates its importance by RDF rank. Hover over a node of interest to open a menu with four options. Click the expand icon to see the links for the chosen node. (You can also expand a node by double-clicking it.)





Click a node to learn more about that resource.



Note: The side panel includes the following:

- labels (`rdfs:label`)
- a short description (`voc:desc`)
- RDF rank

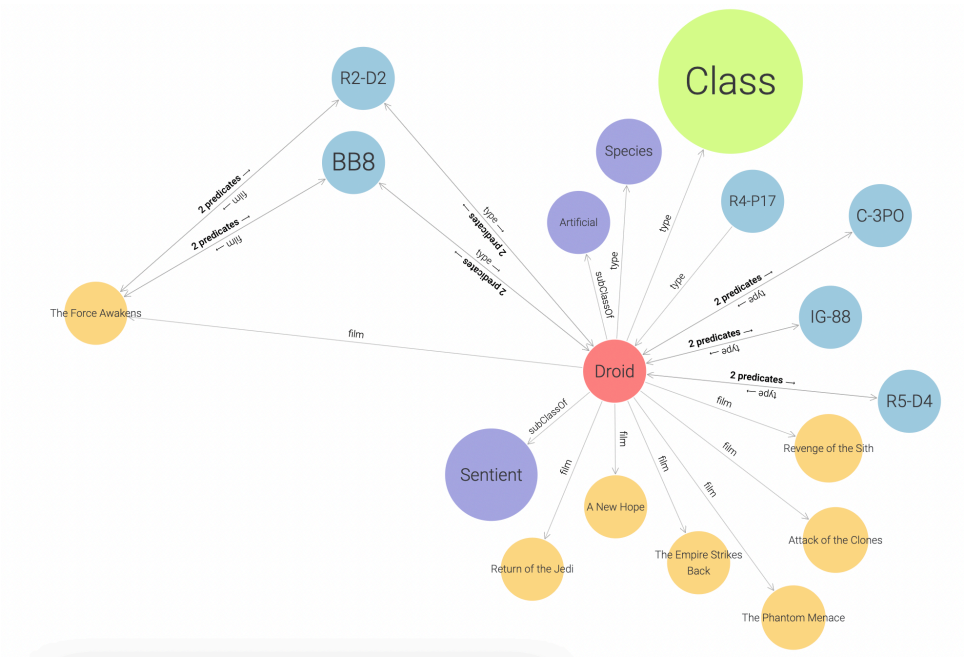
Note that the information in the panel may vary depending on the data you are working with.

Clicking the node again hides the panel.

You can switch between nodes without closing the side panel. Click the new node about which you want to see

more, and the side panel will automatically show the information about it.

Once a node is expanded, you can collapse it. This removes its links and their nodes except those that are also connected to other nodes, as shown in the example below. Collapsing “The Force Awakens” removes all nodes connected to it except “R2-D2” and “BB8”, because they are also linked to “Droid”, which is expanded.

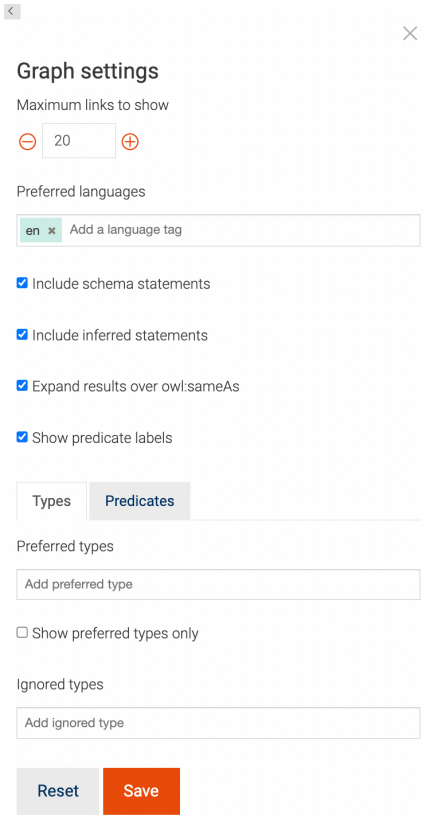


If you are not interested in a node anymore, you can hide it by using the *remove* icon.

The *focus* icon is used to restart the graph with the node of interest. Use carefully, as it resets the state of the graph.

More global actions are available in the menu in the upper-right corner:

- Go back to Visual graph home.
- Search another resource.
- To visually rotate your graph for convenience, use the arrows.
- Pin or unpin all nodes.
- Save this as a named graph.
- Display the *Graph settings* dialog box, shown below.

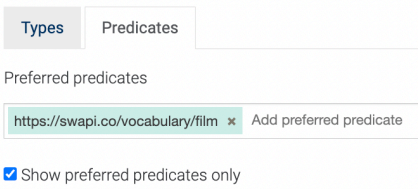


The following settings are available:

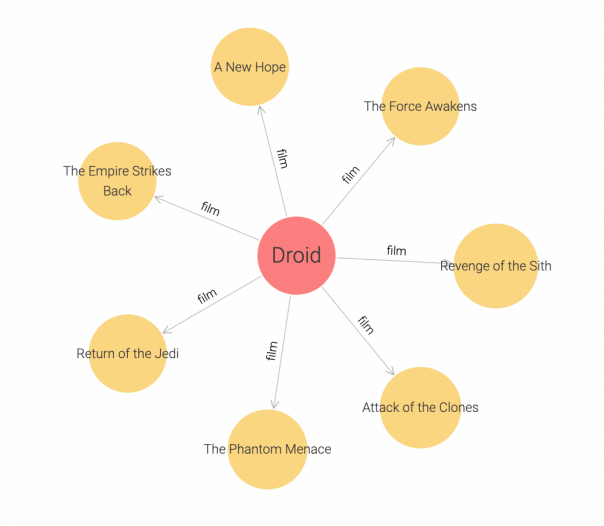
- *Maximum links to show* is the limit of links to use when you expand each node.
- If you have labels in different languages, you can choose which labels to display with preference. The order is important in this case.
- *Include schema statements*
- *Include inferred statements*
- *Expand results over owl:sameAs*
- *Show predicate labels* is an option that you can disable for convenience when you are not interested in the predicates linking the nodes.
- *Preferred* and *Ignored* types and predicates gives you greater control over what is included in the display of expanded nodes. If a preferred type is present, nodes of that type will be shown before all other types, like in the example below. Again, order matters when you have more than one preferred types.

Ignored types are used when you do not want to see instances of some types at all while exploring. The same is valid for predicates. Use full IRIs for types and predicates filters.

For example, add `voc:film` as a preferred predicate and tick the option to see only preferred predicates.

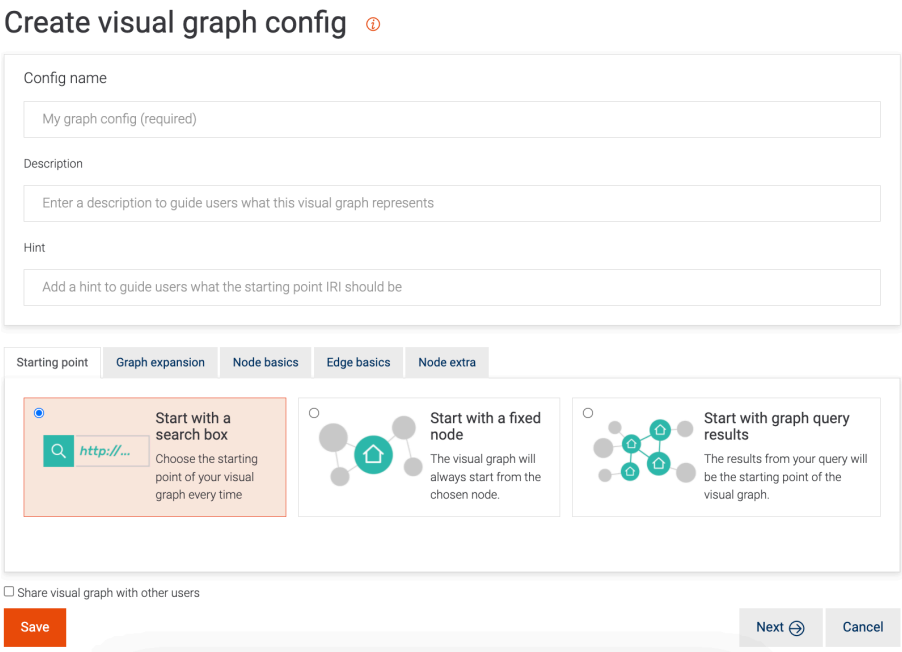


Then click *Save* and see the change:



3.5.3.2 Create your own visual graph

You can create your own custom visual graph by modifying the queries that fetch the graph data. To do this, navigate to *Explore* ‣ *Visual Graph*. In the *Advanced graph* section, click *Create graph config*.



The configuration consists of five queries each stored in its own tab. Each tab includes sample queries to provide a model for your query. Note that some bindings are required.

- Starting point — this is the initial state of your graph.
  - *Search box*: If you want this graph to offer a choice of start resources each time, start with a search box. This is similar to the initial state of the Easy graph.
  - *Fixed resource*: You may want to start exploration with the same resource each time. For example, select http://dbpedia.org/resource/Sofia from the autocomplete input as a start resource so that every time you open the graph you will see Sofia and its connections.
  - *Graph query results*: Visual graph can render a random SPARQL Graph Query result. Each result is a triple that is transformed to a link where the subject and object are shown as nodes and the predicate is a link between them.

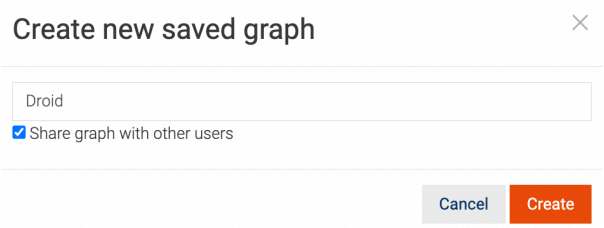
- *Graph expansion*: This is a `CONSTRUCT` query that determines which nodes and edges are added to the graph when the user expands an existing node. The `?node` variable is required and will store the IRI of the expanded node. If empty, the Unfiltered object properties sample query will be used. Each triple from the result is visualized as an edge where the subject and object are nodes and the predicate is the link between them. If new nodes appear in the results, they are added to the graph.
- *Node basics*: This `SELECT` query determines the basic information about a node. Some of that information affects the color and size of the node. This query is executed each time a node is added to the graph to present it correctly. The `?node` variable is required and will store the IRI of the expanded node. It is a `SELECT` query and the following bindings are expected in the results:
  - `?type` determines the color. If missing, all nodes will have the same color.
  - `?label` determines the label of the node. If missing, the IRI's local name will be used.
  - `?comment` determines the description of the node. If missing, no description will be provided.
  - `?rank` determines the size of the node, and must be a real number between 0 and 1. If missing, all nodes will have the same size.
- *Edge basics*: This `SELECT` query sets the `?label` binding that determines the text of the edge. If empty, the edge IRI's local name is used.
- *Node extra*: This `SELECT` query determines the extra properties shown for a node when the info icon is clicked. It should return two bindings: `?property` and `?value`. Results are then shown as a list in the sidebar.

If you leave a query empty, the first sample will be taken as a default. You can execute a query to see some of the results it will produce. Except for the samples, you will also see the queries from the other configurations in case you want to reuse some of them. The combination of all these options provides many ways for your custom visual graph to explore your data.

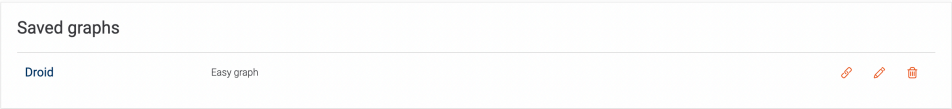
3.5.3.3 Save and share visual graphs

While exploring a graph, you can save a snapshot of the graph state with the *Save* icon in the top right to load it later. The graph configuration you are currently using is also saved, so when you load a saved graph, you can continue exploring with the same configuration.

You can share your saved graphs with other users. When security is ON in the *Setup* ▸ *Users and Access* menu, the system distinguishes between different users. The graphs that you choose to share are only editable by you.



The graphs are located in *Visual graph* ▸ *Saved graphs*. Other users will be able to view them and copy their URLs by clicking the *Get URL to graph* icon. This link will also ask the user loading it whether they want GraphDB to automatically update the active repository to the same one as the saved graph.



When *Users and Access* ▸ *Free Access* is ON, free access users will only see shared graphs and will not be able to save new graphs.

3.5.3.4 Embed visual graphs

GraphDB lets you embed your visual graph by adding the `&embedded` HTTP parameter that hides the Workbench menus (side panel, drop-down, and footer).

The following embedding options are available (substitute `localhost` and the `7200` port number as appropriate):

- Start with a specific resource: `http://localhost:7200/graphs-visualizations?uri=<encoded-iri>&embedded`
- Load a saved state of a specific expanded graph: `http://localhost:7200/graphs-visualizations?saved=<saved-view-id>&embedded`
- Start with a custom graph configuration: `http://localhost:7200/graphs-visualizations?config=<graph-config-id>&embedded`
- Start with graph query results: `http://localhost:7200/graphs-visualizations?query=<encoded-sparql-query>&embedded`

Note: When using embeddings, it's best to run the Workbench in *free access mode*.

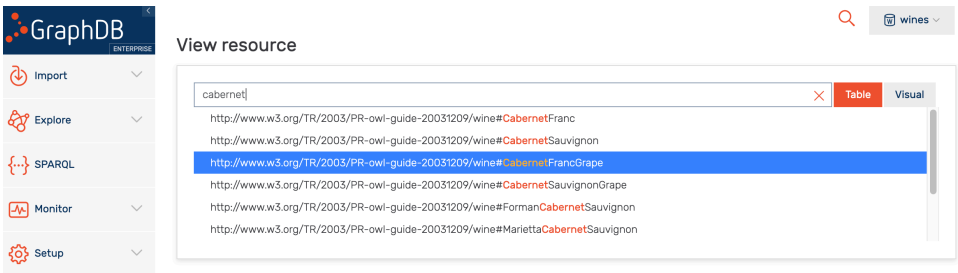
3.5.4 View and edit resources

3.5.4.1 View and add a resource

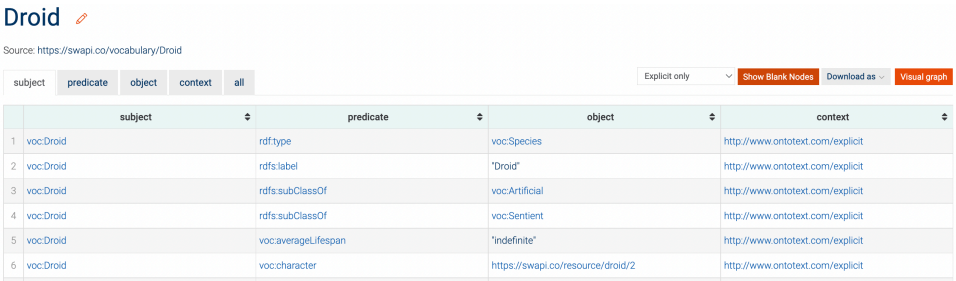
Important: Before using the *View resource* feature, make sure you have enabled the *Autocomplete index* from *Setup* ▸ *Autocomplete*.

To view a resource in the repository, go to the GraphDB home page and start typing in the *Explore* ▸ *View resource* field.

You can also use the *Search RDF resource* icon in the top right. This is visible in all Workbench screens.

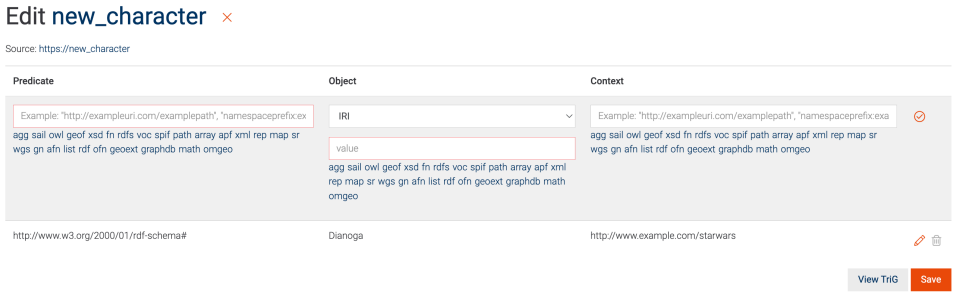


Viewing resources provides an easy way to see triples where a given IRI is the subject, predicate, or object.

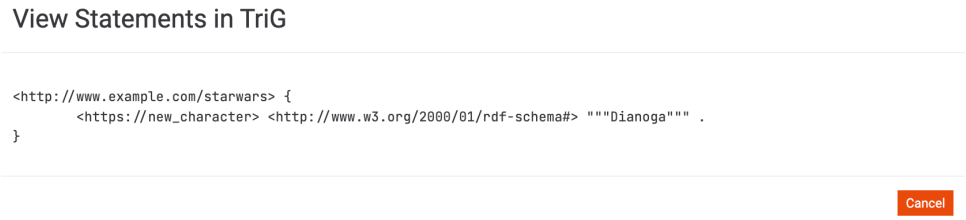


Even when the resource is not in the database, you can still add it from the resource view. Type in the resource IRI and press Enter.

Here, you can use the resource editor to create as many triples as you like. To add a triple, fill in the necessary fields and click on the orange tick on the right. The created triple appears, and the *Predicate*, *Object*, and *Context* fields are empty again for you to insert another triple if you want to do so. You can also edit or delete already created triples.



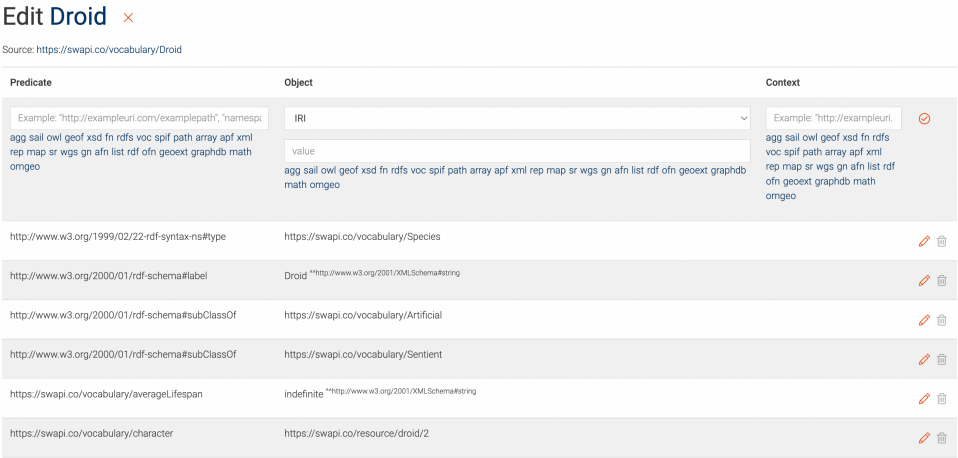
To view the new statements in TriG format, click the *View TriG* button.



When ready, save the new resource to the repository.

3.5.4.2 Edit a resource

Once you use *View resource* to open a resource you can also edit it. Click the edit icon next to the resource namespace and add, change, or delete the properties of this resource.



Note: You cannot change or delete the inferred statements.

3.6 GraphQL

GraphDB also allows you to explore your data using GraphQL — a simple language for querying various data stores. It is based on a strong type system (schema language) that allows objects and query validation, query autocompletion, and query type checking in modern languages such as TypeScript. It is not a replacement for SPARQL as it does not allow arbitrary queries to be issued. Instead, GraphQL is intended to provide a simple facade over complex backend storage systems.

GraphQL queries have a regular hierarchical structure. Each level is serviced (resolved) in sequence, and lower levels cannot influence the results at a higher level. If you want to order or filter by lower-level data, you often need to access the same fields in the higher-level filter clause, and in the lower-level selection clause.

3.6.1 GraphQL security

Since using GraphQL assumes a more limited access to the data, *the GraphDB security mechanism* allows administrators to provide users with access only to the GraphQL environment, without enabling access to the RDF layer. In other words, administrators can create GraphQL-only users who won't be able to use SPARQL, mirroring most real-life use cases of GraphQL.

In addition to the GraphQL-only security check controlled through the User Management screen, a more robust *Role-Based Access Control (RBAC)* mechanism is also available. Whenever a GraphQL query or mutation is invoked, the RBAC mechanism controls which information is returned and which data can be modified in case of a mutation. The mechanism uses *Custom user roles*.

See also:

Migrating GraphQL configurations from Semantic Objects to GraphDB details the important details and steps when migrating from Semantic Objects to GraphDB 11.

3.6.2 GraphQL endpoints

Usage of GraphQL is done through GraphQL endpoints which provide API access to a repository's data. The endpoint works through a generated GraphQL schema which specifies how the repository's classes and properties are treated as GraphQL types and fields. When you create a GraphQL endpoint, you can have GraphDB generate the endpoint from the existing OWL ontologies and SHACL shapes, or use your own GraphQL Schema Shapes Model.

Further down below you can find an example of generating a GraphQL endpoint from a SHACL shape.

See also:

The GraphQL schemas and endpoint management documentation provides a thorough overview of the creation, management and usage of GraphQL endpoints.

3.6.3 Licensing GraphQL

Basic GraphQL support is available in all versions of GraphDB, but some advanced options require an additional GraphQL license:

- *Role Based Access Control (RBAC)*
- Schema validation against data
- GraphQL federation

Note: These restrictions apply on an operational level. Schema parsing, validation and loading will work fine even without a valid license. An operation (query or mutation) is stopped only upon execution and only if any of the advanced features are necessary for its execution.

See also:

For more information on licensing options, check the *Licensing* documentation.

3.6.4 GraphQL Playground

The GraphDB Workbench comes with *GraphQL Playground* — an environment that allows query autocompletion. The GraphQL Playground is built on GraphiQL, a graphical in-browser GraphQL IDE. You can access this interface through *GraphQL* ‣ *GraphQL Playground* in the Workbench menu.

To perform a GraphQL query, you must first select an available *GraphQL endpoint* from the available list of endpoints, found at the top right of the GraphQL Playground interface.

Once you have selected an endpoint from the drop-down menu, you will be able to write and perform GraphQL queries. You can also perform some additional operations:

- Executes the query and displays the results in the tab to the right.
- Automatically formats the GraphQL query, making it easier to read and and work with.
- Merges fragments into query. This only works for in-line fragments.
- Copies the query.

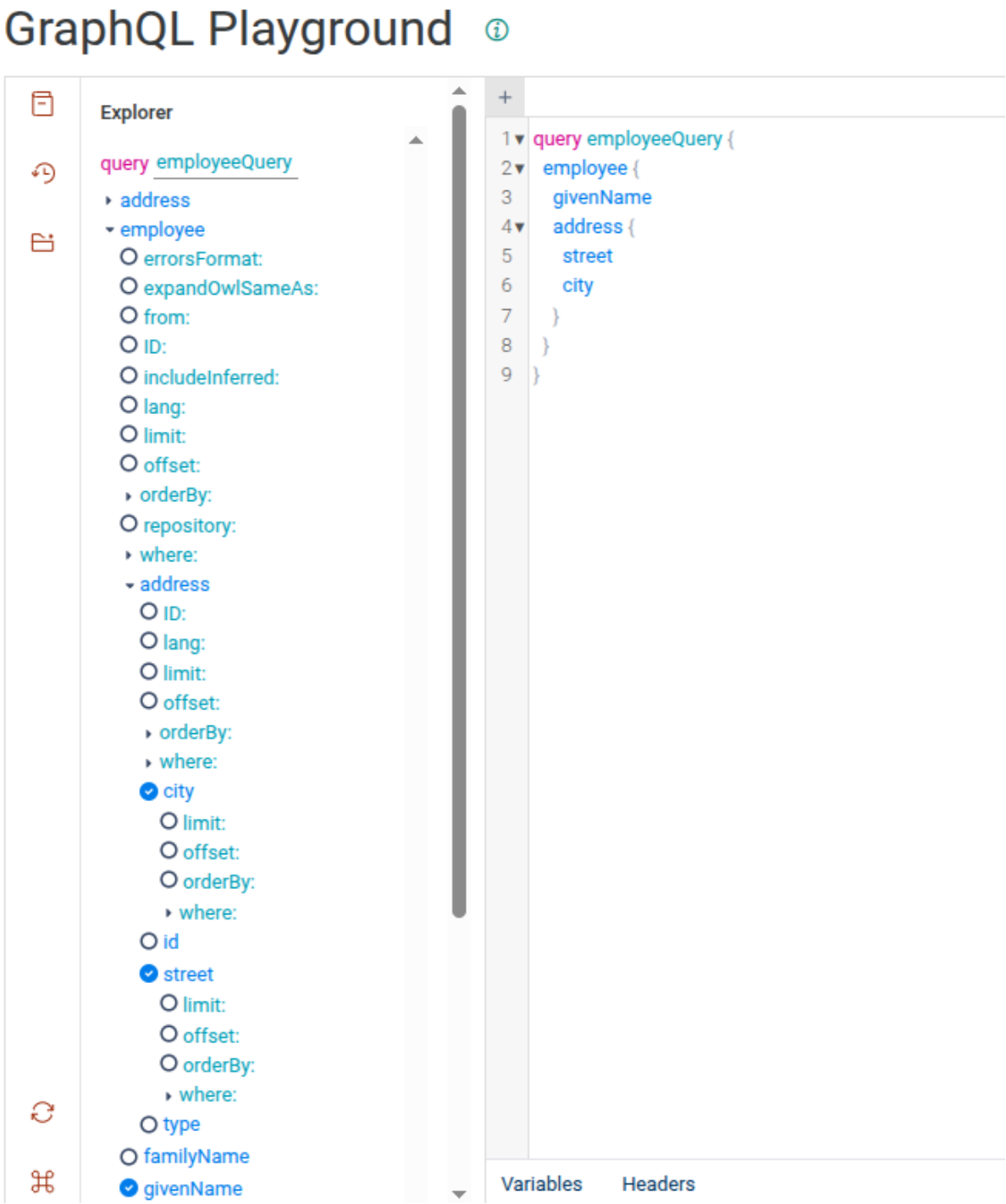
There are several other plugins available in the GraphQL Playground:

- Opens the *Documentation explorer* of your schema. This plugin allows you to browse and explore the schema and its types and fields. This is the default plugin found in GraphiQL, and has mostly been superseded by the *GraphiQL Explorer*, also available in the playground.
- Opens the *History* tab, which shows previously executed queries. You can mark previous queries as *Favorite* for quick access.
- Opens the *GraphiQL Explorer*, an advanced view of the types and fields available in your schema which also allows you to directly construct queries through its interface.

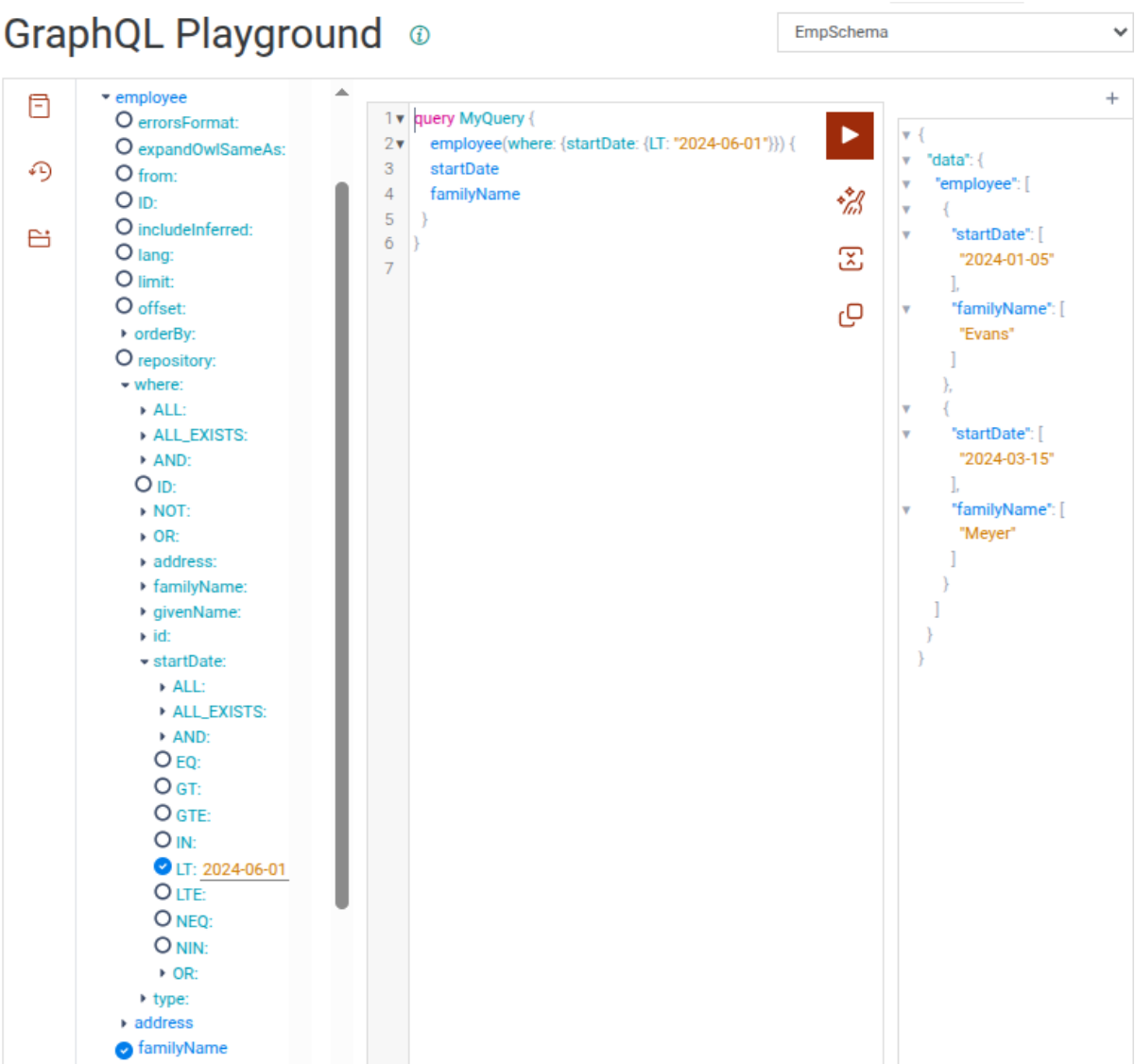
3.6.4.1 Using your GraphQL endpoints with the GraphiQL Explorer

Querying GraphQL endpoints in Workbench

The GraphiQL Explorer can present an intuitive display of more advanced data relationships. For example, in addition to showing the `Employee` and `Address` classes from a schema as top level types in the illustration below, it also shows that the `Employee` type has an `address` field whose value is an instance of the `Address` type with its `city` and `street` fields.



GraphiQL can also show you the operators that you can use for filtering data from this GraphQL endpoint such as `EQ` for equals and `GT` for greater-than. The following shows how the GraphiQL Explorer was used to create a query for all employees with a `startDate` before June 1, 2024, and on the right we can see the result of the query:



Available operators will come from the following list, depending on the type of the field whose value is used as the filter:

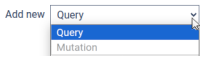
operator	description	example
`ALL`	For a potentially multivalued field, make sure that all meet a certain condition. For example, if an employee may have more than one address, only select them if "Main" is part of the street value for all of the addresses. This will also select employees who have no address street value.	`employee(where: {ALL: {address: {street: {RE: "Main"}}}})`
`ALL_EXISTS`	Like `ALL`, but the field or fields with the value being checked (in the case of the example, the address street value) must exist for that employee.	`employee(where: {ALL_EXISTS: {address: {street: {RE: "Main"}}}})`
`AND`	an array in which all conditions must be true	`employee(where: {AND: [ {address: {city: {EQ: "Springfield"}}} {address: {street: {EQ: "24 Hauptstrasse"}}} ]})`
`EQ`	equal to	`employee(where: {startDate: {EQ: "2024-06-01"}})`
`GT`	greater than	`employee(where: {startDate: {GT: "2024-06-01"}})`
`GTE`	greater than or equal to	`employee(where: {startDate: {GTE: "2024-06-01"}})`
`ID`	matches the specified resource ID; for a GraphQL interface to RDF data, the resource's IRI	`employee(ID: "http://example.com/ns/data#emp2")`
`IN`	in the following array as a member	`employee(where: {familyName: {IN: ["Evans", "Green"]}})`
`IRE`	matches the regular expression (case-insensitive)	`employee(where: {familyName: {IRE: "^ev"}})`
`LT`	less than	`employee(where: {startDate: {LT: "2024-06-01"}})`
`LTE`	less than or equal to	`employee(where: {startDate: {LTE: "2024-06-01"}})`
`NEQ`	not equal to	`employee(where: {startDate: {NEQ: "2024-06-01"}})`
`NIN`	not in the following array as a member	`employee(where: {familyName: {NIN: ["Evans", "Green"]}})`
`NIRE`	doesn't match the regular expression (case-insensitive)	`employee(where: { familyName: {NIRE: "^ev"}})`
`NRE`	doesn't match the regular expression (case-sensitive)	`employee(where: { familyName: {NRE: "^Ev"}})`
`OR`	an array in which at least one of the conditions must be true	`employee(where: {OR: [ {address: {city: {EQ: "Springfield"}}} {address: {street: {EQ: "24 Hauptstrasse"}}} ]})`
`RE`	matches the regular expression (case-sensitive)	`employee(where: { familyName: {RE: "^Ev"}})`

Adding, updating, and deleting data with GraphQL mutations

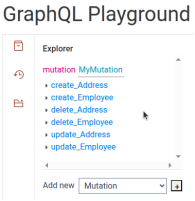
In addition to querying data over a GraphQL endpoint created with GraphDB, as shown in *Creating a GraphQL interface from GraphQL schema shapes* above, the GraphQL schema generated by GraphDB lets you use GraphQL mutations to add, update, and delete data on the server.

Note: The *Create new user* section of the GraphDB documentation describes how you can control GraphQL write access globally and for individual users.

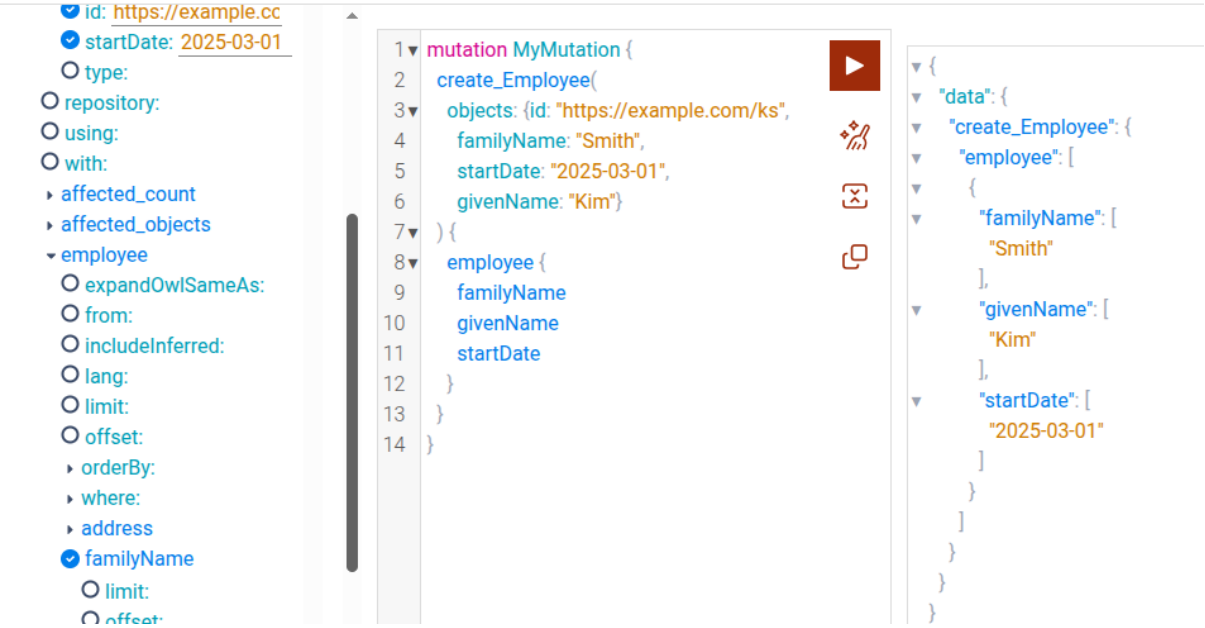
The GraphiQL tool available on GraphDB's GraphQL Playground gives you a visual interface to create both queries and mutations of the data behind your GraphQL endpoint:



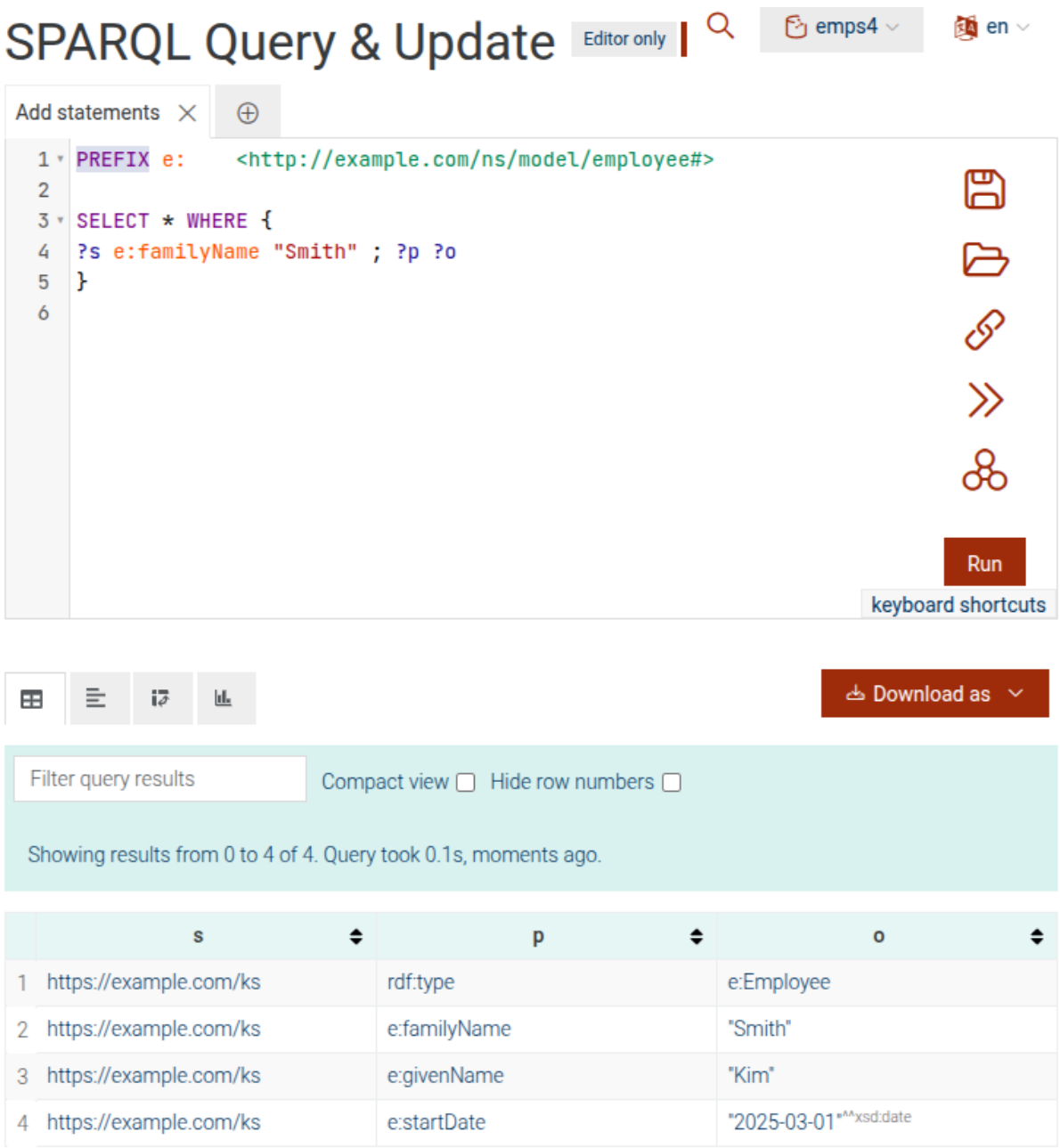
Once you pick *Mutation* from the *Add new* field, GraphiQL offers you a choice of templates to create, delete, or update data for each of the available types:



The following shows a `create_Employee` update operation in GraphiQL that creates a new instance of the `Employee` type by giving it a new `id` value and values for each of its fields. The lower half of the request specifies the data that should be returned so that we can confirm that the instance was created as planned: the values of the newly-created instance. On the right, we see that after executing the operation, the instance was created:



Because GraphDB's GraphQL interface provides access to the RDF data in the underlying repository, the Employee instance created by the update operation is then stored as a set of triples there, as we can see from this SPARQL query:



The `create_Employee` example above specified a hard-coded `id` value to use for the new instance. If you don't specify an ID value for a new instance, the GraphQL schema creation will automatically generate one by appending a unique hexadecimal value to a base URI whose default value is `http://example.org/resource/`. You can reset the base URI of generated IDs by assigning an IRI to the empty prefix. In Turtle, that would look like this:

```
@prefix : <http://custom-iri.com/some/path/> .
```

When generating new IDs, you can use alphanumeric reference identifiers to generate IDs that link from one new object to another. These must be defined in a create operation at root or nested level. In the following, the first part of the query on the left creates an Address instance with "::1" as the ID value. The second part creates an Employee instance with `{ ids: [""::1"]}` as the value of its `address` field. In the results of the mutation operation on the right, we see that the ID generated for the new address instance (ending with "65f8") is the ID of the new employee's address:

```
mutation createEmployeeWithAddress {

  create_Address(objects: { id: "::1",
                  street: "Rue Rouge",
                  city: "Paris"}) {
    address {
      street
      id
    }
  }

  create_Employee(objects: { familyName: "Smith",
                  startDate: "2025-03-01",
                  givenName: "Kim",
                  address: { ids: ["::1"]}}) {
    employee {
      id
      familyName
      address {
        id
      }
    }
  }
}
```

```
{
  "data": {
    "create_Address": {
      "address": [
        {
          "street": [
            "Rue Rouge"
          ],
          "id": "http://example.org/resource/58fbf2a0-4758-5a0e-a1b8-64e0db2265f8"
        }
      ]
    },
    "create_Employee": {
      "employee": [
        {
          "id": "http://example.org/resource/aa86e566-462c-5292-9252-21d397d6552f",
          "familyName": [
            "Smith"
          ],
          "address": [
            {
              "id": "http://example.org/resource/58fbf2a0-4758-5a0e-a1b8-64e0db2265f8"
            }
          ]
        }
      ]
    }
  }
}
```

To update existing data using an update mutation, provide the IDs of the instances you want to modify and specify the fields to be updated along with their new values. The following example updates the `givenName` value of the `Employee` instance that was created above. Note that the ID value is specified inside of square brackets because you can submit a list of IDs to update. Like the mutation that originally created this employee's instance, this update mutation also specifies field values to return, including special `affected_count` and `affected_objects` fields to show us what exactly got affected by the mutation. You can see the results on the right.

```
delete_Employee
update_Address
update_Employee
  errorsFormat:
  expandOwlSameAs:
  graph:
  includeInferred:
  objects*:
    address:
    familyName:
    givenName:
      patch:
      replace: true
      value: "Fran"
    id:
    startDate:
    type:
  repository:
  using:
  where:
    ALL:
    ALL_EXISTS:
    AND:
    ID: https://example.cc
    NOT:
    OR:
    address:
    familyName:
    givenName:
    id:
    startDate:
    type:
```

```
mutation MyMutation {
update_Employee(where: {ID: "https://example.com/ks"},
  objects: {
    givenName: {value: "Fran", replace: true}
  })
{
  employee {
    givenName
    familyName
}
    affected_count {
      kind
      count
    }
    affected_objects {
      kind
      ids
    }
  }
}
```

Variables Headers

```
{
  "data": {
    "update_Employee": {
      "employee": [
        {
          "givenName": [
            "Fran"
          ],
          "familyName": [
            "Smith"
          ]
        }
      ],
      "affected_count": [
        {
          "kind": "Employee",
          "count": 1
        }
      ],
      "affected_objects": [
        {
          "kind": "Employee",
          "ids": [
            "https://example.com/ks"
          ]
        }
      ]
    }
  }
}
```

A delete operation is similar to an update, because the request includes a list of one or more instance IDs to act on, but that's all you need. You can also optionally include the special `affected` fields in the return portion of the request to find out which data was affected. The mutation below deletes the instance that was created and updated in the previous two examples.

```
mutation MyMutation {
delete_Employee(where: {ID: ["https://example.com/ks"]})
    employee {
      id
    }
    affected_count {
      kind
      count
    }
    affected_objects {
      kind
      ids
    }
  }
}
```

```
{
  "data": {
    "delete_Employee": {
      "employee": [
        {
          "id": "https://example.com/ks"
        }
      ],
      "affected_count": [
        {
          "kind": "Employee",
          "count": 1
        }
      ],
      "affected_objects": [
        {
          "kind": "Employee",
          "ids": [
            "https://example.com/ks"
          ]
        }
      ]
    }
  }
}
```

3.6.5 Generating an endpoint from SHACL using the Star Wars dataset

This example uses data describing Star Wars movies, characters, planets and vehicles. It shows how to generate a GraphQL endpoint from SHACL shapes and use it to perform simple queries in the GraphQL Playground.

3.6.5.1 Prerequisites

Before you go through this example, download the `starwars-data.ttl` dataset and import it into a new repository called **star-wars**. Then import the `swapi-shapes-ex1` shapes and import them into a named graph in the same repository.

3.6.5.2 Generate an endpoint from named graph

Once you have imported your data and connected to the **star-wars** repository:

1. Go to *GraphQL* ▸ *Endpoint Management*, then select *New endpoint*
2. Select *OWL Ontologies/SHACL Shapes*:
   - Under *Endpoint ID*, provide a unique name for the endpoint. This example uses **star-wars**.
   - Under *Vocabulary prefix*, select **voc**.
   - You can also provide a short *Endpoint description*.
3. Choose *Select one or more graphs* to move the graph from *Available (not included)* to *Included for generation*.
4. Click on ⊕ next to the name of your graph.

Tip: Since the repository should contain only one named graph at this point, you can also click on *Add all* instead.

5. Select *Next* to move to the next screen. For the purposes of this example, leave all configurations on that page at their default values.
6. Select *Next*, then *Create endpoint*.

Once GraphDB finishes creating the endpoint, you can either navigate to *GraphQL* ‣ *GraphQL Playground*, or directly select the name of the repository to load it into the GraphQL Playground.

See also:

- *GraphQL schemas and endpoint management* explains the generation of endpoints in detail, including the generation of endpoints from GraphQL schemas.

3.6.5.3 Executing simple queries in the GraphQL Playground

Once you have created your endpoint, you can use it to explore your data using GraphQL queries. Navigate to *GraphQL* ‣ *GraphQL Playground*, and make sure that the **star-wars** endpoint is selected from the top-down menu at the top right of the GraphQL Playground.

Note: You can explore the GraphQL schema through the *Documentation Explorer* and *GraphiQL Explorer*. Note that the types and their properties in your endpoint correspond to the SHACL shapes.

First, let's get a list of all characters in Star Wars, ordered by height in descending order. Paste this query into the Query editor, then select ▶ to execute the query:

```
query charactersOrderedByHeight {
  character(orderBy: {height: DESC}) {
    gender
    label {
      value
    }
    id
    height
  }
}
```

This query should return a result that begins with:

```
{
  "data": {
    "character": [
      {
        "gender": "male",
        "label": [
          {
            "value": "Yarael Poof"
          }
        ],
        "id": "https://swapi.co/resource/quermian/57",
        "height": "264.0"
      },
      {
        "gender": "male",
        "label": [
          {
            "value": "Tarfful"
```

(continues on next page)

(continued from previous page)

```
          }
        ],
        "id": "https://swapi.co/resource/wookiee/80",
        "height": "234.0"
      },
      {
        "gender": "male",
        "label": [
          {
            "value": "Lama Su"
          }
        ],
        "id": "https://swapi.co/resource/kaminoan/72",
        "height": "229.0"
      },

      ...
```

However, the list of all characters in this repository is rather extensive. If you wish, you can instead query for a smaller set of characters — such as a query that will return only the characters that appear in **Star Wars Episode IV: A New Hope**, and limit the results to the first five characters:

```
query allCharactersInANewHope {
  character(limit: 5, where: {film: {ID: "https://swapi.co/resource/film/1"}}) {
    id
    label {
      value
    }
  }
}
```

Which should return the following:

```
{
  "data": {
    "character": [
      {
        "id": "https://swapi.co/resource/droid/2",
        "label": [
          {
            "value": "C-3PO"
          }
        ]
      },
      {
        "id": "https://swapi.co/resource/droid/3",
        "label": [
          {
            "value": "R2-D2"
          }
        ]
      },
      {
        "id": "https://swapi.co/resource/droid/8",
        "label": [
          {
            "value": "R5-D4"
          }
        ]
      },
      {
```

(continues on next page)

(continued from previous page)

```
        "id": "https://swapi.co/resource/wookiee/13",
        "label": [
          {
            "value": "Chewbacca"
          }
        ]
      },
      {
        "id": "https://swapi.co/resource/rodian/15",
        "label": [
          {
            "value": "Greedo"
          }
        ]
      }
    ]
  }
}
```

3.7 GraphQL schemas and endpoint management

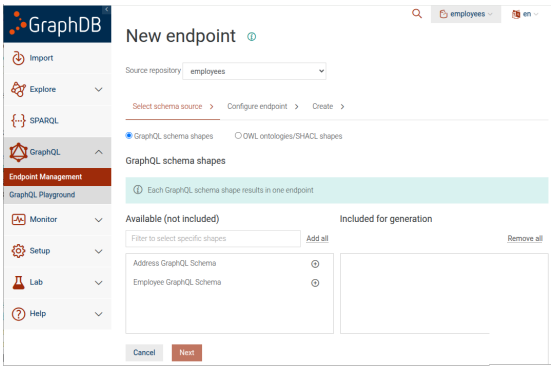
GraphDB lets you create *GraphQL* endpoints that provide API access to a GraphDB repository's data, treating the repository's classes and properties as GraphQL types and fields. To do this, GraphDB generates a GraphQL schema that specifies the structure of this interface.

There two are ways to provide input for this process: a *GraphQL Schema Shapes model* or a set of *OWL ontologies and SHACL shapes*. See the *Creating GraphQL endpoints* section for an overview of the two.

After you have used one of these approaches to create a GraphQL endpoint for one of your repositories, the GraphiQL Explorer tool that is built in to Workbench's GraphQL Playground gives you a visual way to explore the new API. It lets you create queries and also mutations that can add to, delete, and modify the data. Queries and mutations can then be programatically sent to the endpoint using the *GraphDB REST API*.

3.7.1 Creating GraphQL endpoints

From Workbench, selecting *GraphQL* ▸ *Endpoint Management* takes you to the *Endpoint Management* screen. The *New endpoint* button on this screen takes you to the *New endpoint* screen.



The *Source repository* for you new endpoint will be set to the currently connected one, but you can use the dropdown list near the top of this screen to select another one.

Underneath that, select the radio button indicating whether you want to create the schema from *GraphQL Schema shapes* or *OWL Ontologies/SHACL shapes*. Each of these offers certain advantages depending on the input you are using to configure the new schema. Select one of the links in the following list for details on using that method:

- *GraphQL schema shapes*: Describe your data structures with a GraphQL model that uses the GraphQL Data Shapes vocabulary to specify the high-level structure and *SHACL* to specify details about the types and fields to make available via the API. This method is best when you want to have more control over the GraphQL schema that gets generated.
- *OWL ontologies/SHACL shapes*: Identify existing RDFS schemas, OWL ontologies, and SHACL shapes to use as input for GraphDB's API generator. This method is best when you want to quickly generate a GraphQL schema from models that you already have.

3.7.1.1 Creating a GraphQL interface from GraphQL schema shapes

When you create a GraphQL endpoint in Workbench by selecting *GraphQL* ▸ *Endpoint Management* and then *New endpoint*, the *GraphQL schema shapes* option lets you create one using a GraphQL Schema Shapes model. The creation and tuning of this model lets you customize the appearance and behavior of the GraphQL API and endpoint that you are creating.

To do this, the GraphQL Data Shapes vocabulary provides classes and properties that you can use to define a GraphQL Schema Shapes model. GraphDB can then use this model to generate a GraphQL schema, endpoint, and API.

Preparing the model to load

A GraphQL Schema Shapes model consists of:

- An instance of `graphql:Schema` from that vocabulary.
- Properties of that instance to describe the schema's structure.
- Any supporting shapes, RDFS, and OWL classes necessary to describe the classes and properties that will be exposed as types and fields in the GraphQL API being generated.

The GraphQL Schema Shapes model is typically stored in the repository with the instance data that it describes so that once the model is used to generate a GraphQL schema and endpoint, GraphQL queries will return instance data from that repository via the endpoint whose API is defined by this model.

You can use multiple GraphQL Schema Shapes models to generate multiple GraphQL APIs for a single repository, with each API providing a different view of the repository's data. When you are connected to a repository that has any of these models defined, selecting *GraphQL* ▸ *Endpoint Management* in Workbench and then *New endpoint* will show you any `graphql:Schema` shapes defined for that repository in the *Available* list.

The following shows an example GraphQL Schema Shapes model. After declaring `e:EmpSchema` as an instance of `graphql:Schema`, it has an `rdfs:label` value and two `graphql:publicShape` properties to point to shapes that represent data classes in that repository. The rest of the schema is all SHACL statements to describe those classes and their relationship.

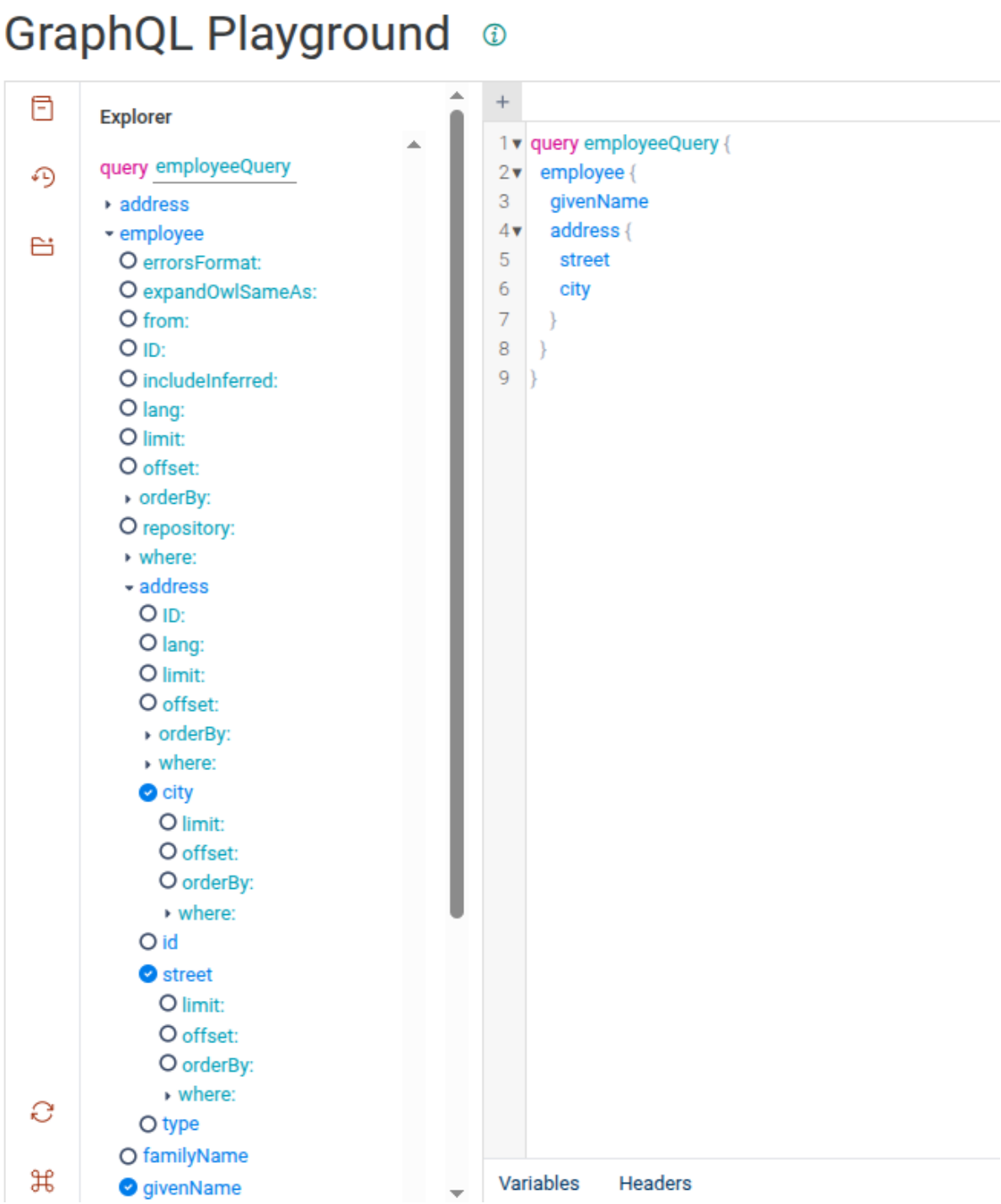
```
@prefix e:       <http://example.com/ns/model/employee#> .
@prefix graphql: <http://datashapes.org/graphql#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh:      <http://www.w3.org/ns/shacl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

e:EmpSchema a graphql:Schema;
            rdfs:label "Employee GraphQL Schema" ;
            graphql:publicShape e:EmployeeShape ;
            graphql:publicShape e:AddressShape .

e:EmployeeShape a sh:NodeShape ;
            sh:targetClass e:Employee ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:datatype  xsd:string ;
```

(continues on next page)

(continued from previous page)

```
                        sh:path      e:familyName
                      ] ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:datatype  xsd:string ;
                          sh:path      e:givenName
                      ] ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:datatype  xsd:date ;
                          sh:path      e:startDate
                      ] ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:class     e:Address ;
                          sh:path      e:address
                      ] .

e:AddressShape a sh:NodeShape ;
            sh:targetClass e:Address ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:datatype  xsd:string ;
                          sh:path      e:street
                      ] ;
            sh:property [ rdf:type     sh:PropertyShape ;
                          sh:datatype  xsd:string ;
                          sh:path      e:city
                      ] .
```

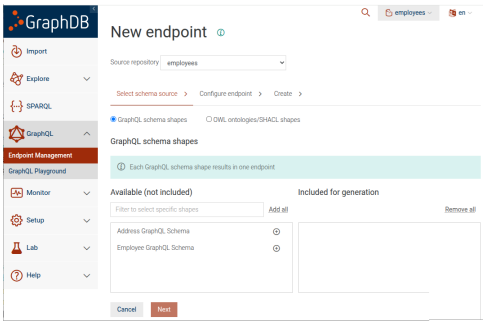
This example uses a limited range of SHACL features, but you can use additional ones to create a more sophisticated model. As described later in this chapter, the GraphQL Data Shapes vocabulary includes additional constructs that you can add to this model to define more details about the generated GraphQL shapes. Still, with a minimal GraphQL Schema Shapes model like the one shown above, you can generate a GraphQL schema that looks like the following when viewed with the GraphiQL Explorer tool that is included with Workbench:



See *Querying GraphQL endpoints in Workbench* for more on using this tool.

Loading the model and creating the GraphQL schema

If your currently connected repository has a *GraphQL Schema Shapes model* in it, then when you *create a new GraphQL endpoint* and select *GraphQL schema shapes* on the *New endpoint* screen, you will see a GraphQL schema shape listed under *Available (not included)* for each `graphql:Schema` defined in repository's data. Below, we see two listed there:



Clicking the *Add the shape to the list of included shapes* icon ⊕ to the right of any available GraphQL Schema Shape in that list will move it over to the *Included for generation* list. Once you have moved all the shapes you want to that list, click *Next* at the bottom of the screen to advance to the *Configure endpoint* screen.

Note: When creating *Creating a GraphQL interface from OWL ontologies and SHACL shapes*, specifying the interface's ID is an important part of its configuration. When creating a *GraphQL schema shapes* one, however, the ID is automatically created from the local part of the IRI identifying the `graphql:Schema` instance used to define the schema. So, if your `graphql:Schema` instance has the IRI `http://example.com/ns/model/employee/EmpSchema`, its interface ID within the repository where it is stored would be "EmpSchema".

This screen includes an *Advanced settings* toggle switch that display additional settings. The following shows a complete list of the regular and advanced parameters grouped in the categories used there:

Note: These settings can also be set using the predicates listed under *GraphQL schema generation configuration options* .

Generic settings

- *Default String Mode*: the default property range when the range is not explicitly defined. Choose from string, langString, or stringOrLangString.
- *Use GraphQL Unions*: whether to `sh:or` together multiple specified types or to just pick one.
- *Force Interface Types*: whether parent types should always be interfaces or split between interface and subclass.
- *Default Maximum Cardinality*: an integer number specifying the maximum cardinality or the string "inf" (the default) to indicate no limit.
- *Use SHACL Property Labels*: whether to use `sh:PropertyShape` labels and descriptions instead of `sh:path` labels and descriptions.

Data Cleanup

- *Remove Unknown Ranges*: whether to remove properties with unknown ranges.
- *Clean Invalid InverseOf Relations*: whether to remove invalid `sh:inverseOf` properties.
- *Include Deactivated Shapes*: whether to include deactivated SHACL shapes in the schema.

Enterprise settings

- *Read RBAC Rules*: whether to read RBAC rules and apply them to the API access. If this is not checked and security is enabled and you do not have a license for extended GraphQL support, the endpoint will not be accessible. (If you do not have a proper license loaded, you would not be able to use the endpoint anyway.)

All of the parameters above have default settings, so if those all look fine to you then you can just click the *Next* button to continue.

The *Configure endpoint* screen includes a *Create endpoint* button. Once you click it, GraphDB will generate an endpoint for each GraphQL schema shape that it found and list them as links to a *GraphQL Playground* where you can use GraphiQL to generate queries and mutations to execute with your new endpoint. Read more about this at *Using your GraphQL endpoints with the GraphiQL Explorer*.

Customizing data access

In the sample GraphQL Schema Shapes model above, both of its classes were declared as `graphql:publicShape` shapes. The following list of properties from the datashapes vocabulary includes other options that you can use to control the visibility of different parts of your model in the GraphQL endpoint API.

Note: See *GraphQL schema generation configuration options* for background on the ontology that defines the `so:` properties.

- `graphql:Schema`: an instance of this type provides the IRI identifier for the GraphQL Schema Shapes model.
- `graphql:name`: when assigned to the `graphql:Schema` instance, this becomes the endpoint name used in Workbench. When assigned to a node shape, it becomes the name of the type generated from that nodeshape in the GraphQL endpoint and will be displayed as that type's name in GraphiQL. When assigned to a property shape, it becomes the name of the field generated from that property shape and will be displayed as that field's name in GraphiQL.
- `graphql:publicShape`: As shown in the example above, this property of a `graphql:Schema` instance identifies which declared SHACL shapes should be part of that instance's GraphQL shapes model. A public shape can be queried via the GraphQL endpoint without restrictions. If the supertypes of the referenced type of a shape are not added explicitly in the model, the parent properties will be included in the type.
- `graphql:publicClass`: This property of a `graphql:Schema` instance declares declares which classes will be exposed as types in the GraphQL API. These classes and their subclasses will be included in the GraphQL schema and can be queried without restrictions.
- `graphql:privateShape`: This alternative to `graphql:publicShape` indicates that a shape should not be exposed as part of the GraphQL API being defined. This can be useful when defining multiple APIs to the same dataset. If a relation has one of these types as range then the property will be changed to a range of IRI. To hide the property itself, use `so:privateProperty`, which is described below.
- `graphql:protectedShape`: This alternative to `graphql:publicShape` indicates that a shape's type cannot be queried by itself — that is, no queries or mutations can directly read or write instances of this type — but, these operations can be done via relations coming from public types or other protected ones. For example, if `Address` in the example above was a protected shape, it would not be shown in GraphiQL as a top-level type but instances of this type would still be accessible when `Employee` instances had `address` values pointing to them.
- `graphql:protectedClass`: This property of a `graphql:Schema` instance declares that a class will not be exposed as a type in the GraphQL API. This class and its subtypes will be included in the GraphQL schema but cannot be queried directly and can only be accessed via relation.
- `graphql:defaultPrefix`: Points at a `sh:prefix` value and `sh:namespace` IRI value to indicate which namespace should be treated as the default. For most type and field names generated as part of the GraphQL interface, the relevant prefix is prepended with an underscore — for example, a `d:givenName` property would be shown as `d_givenName`. This will not be done for the prefix and namespace identified with the `graphql:defaultPrefix` property. If the prefix named here was `d:` along with its corresponding IRI, the example field name would be `givenName`. This value is also used to initialize the vocabulary IRI for the schema.

- `graphql:isInterface` or `dash:abstract`: If set to true for a node shape or type, then that node shape or type should become a GraphQL interface.
- `graphql:uriTemplate`: Can be used to attach a URI template to node shapes. This provides an alternative to the `so:pattern` predicate.
- `so:privateProperty`: specifies a list of properties that should be hidden from the GraphQL schema.
- `so:protectedProperty`: specifies a list of relations that should not expose the range of the related property. These properties will get a range of IRI as the default.
- `so:publicNamespace`: The namespace or namespaces of node shapes that shall be published by the GraphQL schema. This will use the values of `sh:namespace` of the linked prefix declarations. This can be used to import other GraphQL models based on their `graphql:defaultPrefix` definition. In the following example, when a GraphQL API is generated for `test:MySchema1` it will include the definitions of the `test:MySchema2` as it matches the `graphql:publicNamespace/sh:namespace` value to the `test:MySchema2` `graphql:defaultPrefix/sh:namespace` value.

```
@prefix test: <http://example.org/test#> .
@prefix test2: <http://example.org/test2#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix graphql: <http://datashapes.org/graphql#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

test:MySchema1 a graphql:Schema ;
  graphql:defaultPrefix [
        sh:prefix "test" ;
        sh:namespace "http://example.org/test#"
  ];
  graphql:publicNamespace [
        sh:namespace "http://example.org/test2#"
  ].

test:MySchema2 a graphql:Schema ;
  graphql:defaultPrefix [
        sh:prefix "test2" ;
        sh:namespace "http://example.org/test2#"
        ];
  graphql:publicShape test2:MyShape.

  test2:MyShape a owl:Class, sh:NodeShape ;
  sh:property [
        sh:path rdfs:label ;
        sh:datatype xsd:String ;
        ].
```

Here is the resulting model shown in YAML syntax:

```
id: /soml/MySchema1
prefixes:
  rdfs: http://www.w3.org/2000/01/rdf-schema#
  test: http://example.org/test#
  test2: http://example.org/test2#
specialPrefixes:
  vocab_prefix: test
  vocab_iri: http://example.org/test#
objects:
  test2:MyShape:
    type: test2:MyShape
    props:
      rdfs:label:
        kind: literal
        max: inf
```

(continues on next page)

(continued from previous page)

```
        range: string
        rdfProp: rdfs:label
```

Available SHACL constraints

The following SHACL constraints are available to specify more details about the type and field data made available with the GraphQL interface:

- sh:class
- sh:datatype
- sh:deactivated
- sh:defaultValue
- sh:description
- sh:equals
- sh:in
- sh:hasValue
- sh:languageIn
- sh:maxCount
- sh:maxExclusive
- sh:maxInclusive
- sh:maxLength
- sh:minCount
- sh:minExclusive
- sh:minInclusive
- sh:minLength
- sh:name
- sh:nodeKind
- sh:or
- sh:property
- sh:targetClass or dash:applicableToClass
- sh:uniqueLang
- sh:order

The `so:sparqlTemplate` property lets you define SPARQL templates to represent the property path. This has two forms, both demonstrated here:

```
@prefix so:      <http://www.ontotext.com/semantic-object#> .
@prefix graphql: <http://datashapes.org/graphql#> .
@prefix sh:      <http://www.w3.org/ns/shacl#> .
@prefix voc:     <http://example.com/voc#> .

voc:MyShape a sh:NodeShape ;
  sh:property [
    graphql:name "myTemplateProperty1" ;
    sh:path [
```

(continues on next page)

(continued from previous page)

```
      so:sparqlTemplate "?_subject .. ?_object" ;
      so:restrictive true #/ default is false
    ] ;
    sh:class voc:Person ;
  ] .

voc:MyShape a sh:NodeShape ;
  sh:property [
    graphql:name "myTemplateProperty2" ;
    so:sparqlTemplate "?_subject .. ?_object" ;
    sh:datatype xsd:int ;
```

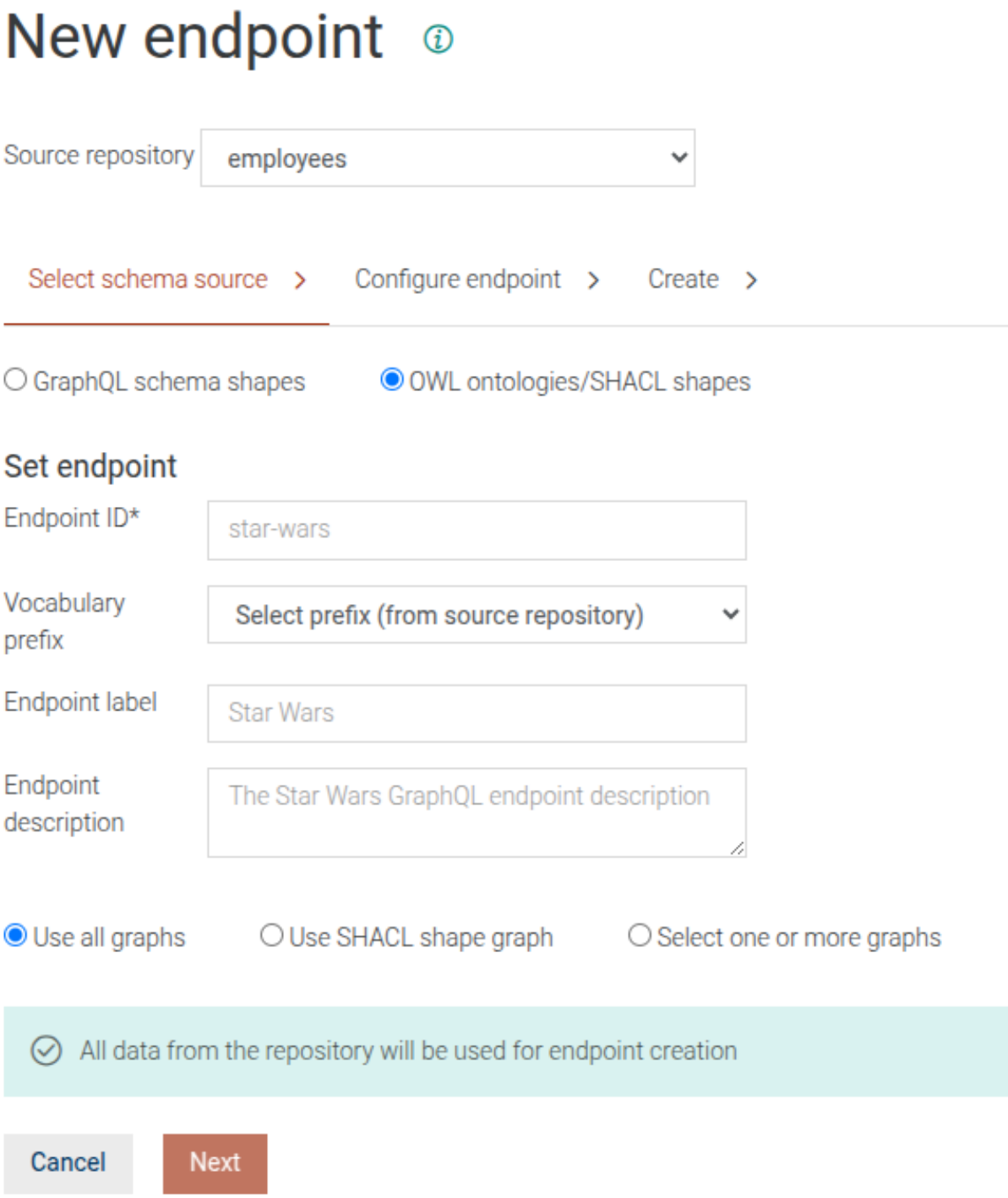
The full form lets you set the template as restrictive. This prevents the SPARQL generation from wrapping the given property in optional clauses when querying. This forces the results to be filtered based on the template matching. This is mainly used when querying GraphDB connectors and results need to be filtered by the query itself.

3.7.1.2 Creating a GraphQL interface from OWL ontologies and SHACL shapes

Without defining a GraphQL Schema Shapes model, you can still have GraphDB generate a GraphQL API and endpoint using existing RDFS and OWL modeling and SHACL shapes. When doing this, keep the following in mind about the models that you provide as input:

- If you have only RDFS and base OWL, then the GraphQL properties are assigned based on domain and range predicates used in the models as well as schema:domainIncludes and schema:rangeIncludes statements.
- If you have OWL restrictions, the model will be based on them plus the constructs listed above.
- If SHACL statements are included, they will have higher priority then the rest. They will still be read but would only affect the model if the `sh:targetClass` does not map to the same `owl:Class` or `rdfs:Class`.

To create this style of GraphQL interface, after selecting *GraphQL ▸ Endpoint Management* and then *New endpoint*, click the *OWL ontologies/SHACL shapes* radio button. This displays a screen where you set general information about the endpoint:



- *Endpoint ID*: the system ID for the endpoint. This is required and will be used in, for example, HTTP API calls that reference this endpoint. This is the only required field on this page.
- *Vocabulary prefix*: the prefix that corresponds to the vocabulary IRI. This is not added to ordinary properties in the GraphQL schema . For example, if `prop` is defined as `pfx:prop`, it will be visualized simply as `prop`. However, a reserved word such as `name` must be defined as `pfx:name` to avoid collision with the predefined property, and will be visualized as `pfx_name`. The same applies to the reserved words `id` and `type`.
- *Endpoint label*: a human-readable name for the endpoint.
- *Endpoint description*: A longer description of the endpoint.

A choice of radio buttons at the bottom of the screen lets you designate which of the repository's graphs to use when generating the GraphQL schema:

- *Use all graphs*: use all the graphs in the repository.
- *Use SHACL shape graph*: use the graph designated for SHACL validation. The default is the `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph` named graph, but you can assign a different name to the SHACL validation graph when you *Enable SHACL validation* while *Creating and configuring a SHACL repository*.
- *Select one or more graphs*: this displays the repository's named graphs so that you can select which of them you would like included to expose in the GraphQL endpoint being created.

Clicking this screen's *Next* button takes you to the *New endpoint* screen.

This screen includes an *Advanced settings* toggle switch that display additional settings. The following shows a complete list of the regular and advanced parameters grouped in the categories used there:

Note: These settings can also be set using the predicates listed under *GraphQL schema generation configuration options* .

Generic settings

- *Default String Mode*: the default property range when the range is not explicitly defined. Choose from string, langString, or stringOrLangString.
- *Use GraphQL Unions*: whether to `sh:or` together multiple specified types or to just pick one.
- *Force Interface Types*: whether parent types should always be interfaces or split between interface and subclass.
- *Default Maximum Cardinality*: an integer number specifying the maximum cardinality or the string "inf" (the default) to indicate no limit.
- *Use SHACL Property Labels*: whether to use `sh:PropertyShape` labels and descriptions instead of `sh:path` labels and descriptions.

Data Cleanup

- *Remove Unknown Ranges*: whether to remove properties with unknown ranges.
- *Clean Invalid InverseOf Relations*: whether to remove invalid `sh:inverseOf` properties.
- *Include Deactivated Shapes*: whether to include deactivated SHACL shapes in the schema.

Enterprise settings

- *Read RBAC Rules*: whether to read RBAC rules and apply them to the API access. If this is not checked and security is enabled and you do not have a license for extended GraphQL support, the endpoint will not be accessible. (If you do not have a proper license loaded, you would not be able to use the endpoint anyway.)

All of the parameters above have default settings, so if those all look fine to you then you can just click the *Next* button to continue.

The next screen includes a *Create endpoint* button. Once you click it, GraphDB will generate an endpoint and provide a link to a *GraphQL Playground* where you can use GraphiQL to generate queries and mutations to execute with your new endpoint. Read more about this at *Using your GraphQL endpoints with the GraphiQL Explorer*.

3.7.2 GraphQL schema generation configuration options

In the RDF provided as input to the GraphQL schema generator, the following properties give you additional customization options. These are from the Semantic Object Modeling Language (SOML) ontology, which uses the prefix `so:` to represent the `http://www.ontotext.com/semantic-object/` namespace. You can find this vocabulary at `soml-ontology.ttl`.

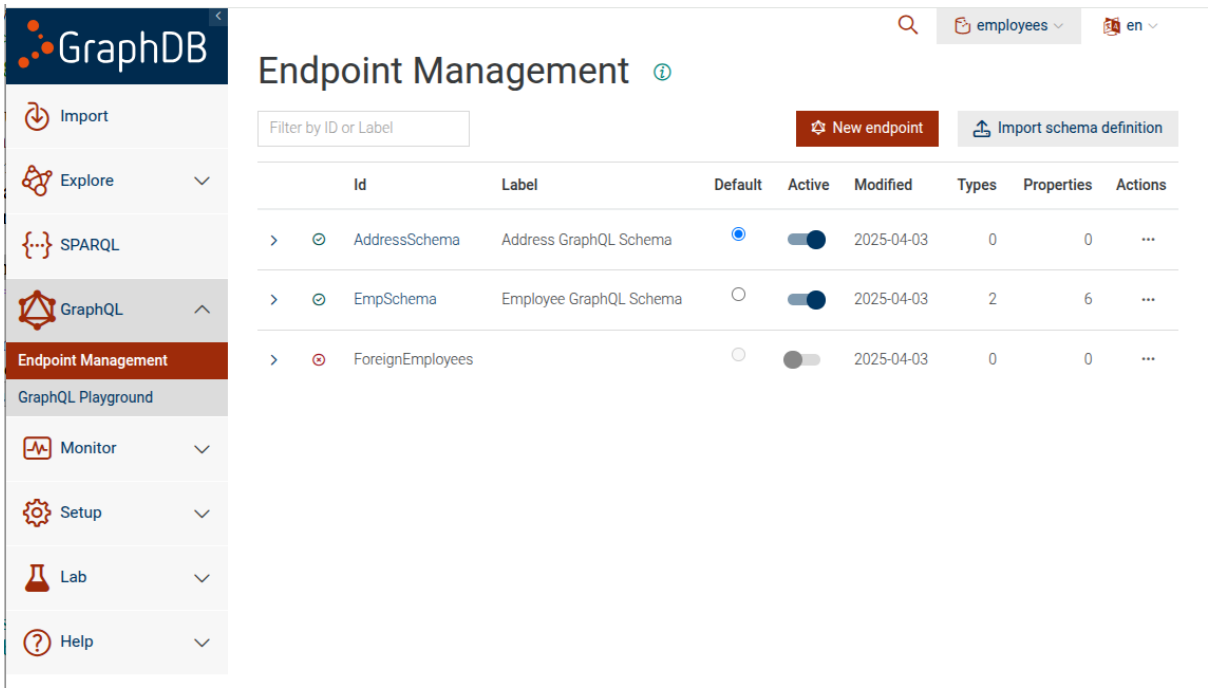
You can set these options for a given GraphQL Schema Shapes model by including a triple like the following in the definition of that model:

```
ex:myGraphQLSchema so:useUnions true .
```

Configuration predicate	Possible values	Default value	Description
`so:readShaclPropertyLabels`	`true` or `false`	`false`	If enabled, the SHACL property labels will have higher priority than the `rdfs:label` values of the predicates that `sh:path` points to.
`so:includeDeactivatedShapes`	`true` or `false`	`false`	Whether the parser schema should honor the `sh:deactivated` predicate and exclude the node and property shapes marked as deactivated. The deactivated shapes will be excluded by default.
`so:cleanInvalidInverseOf`	`true` or `false`	`true`	Whether additional post processing should be done to clean invalid `owl:inverseOf` relations.
`so:useUnions`	`true` or `false`	`true`	Whether the converter should try to use GraphQL unions where possible. This applies when an object property's range has multiple values or the `sh:or` boolean operator is used with more than one type. Note that literal properties with multiple ranges will produce a custom union scalar type for the particular case.
`so:defaultMaxCardinality`	Int number, "inf" or "unbounded"	"inf"	The default maximum cardinality setting to use when not defined explicitly. This determines whether all properties are multi- or single-valued by default.
`so:forceInterfaceTypes`	`true` or `false`	`false`	When enabled, the parent types will always be pure interfaces without concrete superclasses unless explicitly defined as not being interfaces by having the predicates `graphql:isInterface` or `dash:abstract` set to `false`. If the configuration is set to `false`, then the parent types will be threaded as superclasses and a parent interface and concrete instance of the type will be created as well, unless the type is explicitly defined as an interface by having one of the predicates `graphql:isInterface` or `dash:abstract` set to `true`.
`so:removeUnknownRanges`	`true` or `false`	`true`	When enabled, the invalid ranges of the model will be replaced with IRI.
`so:readRBACRules`	`true` or `false`	`true`	Whether the generator should read the RBAC definitions from the model or not. If they are read and the GraphDB license does not include the GraphQL extended capability then the generated endpoint will be restricted.

3.7.3 Managing GraphQL endpoints

Select *GraphQL* ‣ *Endpoint Management* to display the screen that lists the existing GraphQL endpoints for the currently active repository. The example below shows three endpoints stored with an "employees" repository.



Above the table with the endpoint list the screen has the following:

- A *Filter by ID or Label* field in the upper left lets you set the table to only display rows whose *ID* or *Label* values match a string that you enter.
- A *New endpoint* button starts the wizard that leads you through the creation of a new endpoint. See *Creating GraphQL endpoints* for more on this.
- The *Import schema definition file* button in the upper right imports a schema definition file. These files are typically created by exporting them from another schema with the *Export schema definition* choice of each table row's *Actions* ··· menu.

The *Endpoint Management* screen's table has the following information about each endpoint:

- ›: display a longer description of the endpoint if that was includes with the schema definition.
- ⊙: display the creation report from when this endpoint was generated. This is displayed in a dialog box that lets you delete the report or download it as JSON.
- *Id*: the ID assigned to the endpoint when it was created. This will be used in, for example, HTTP API calls that reference this endpoint.
- *Label*: a name that you can assign upon creation that is typically more human-readable than the ID.
- *Default*: Whether this is the default endpoint to use when an operation such as an HTTP API call uses an endpoint from this repository without specifying which one. Clicking one of these circles assigns that circle's endpoint to be the active one, as indicated by the blue filled-in circle.
- *Active*: Whether this endpoint is currently active. Each row includes a toggle button that lets you turn this on or off.
- *Modified*: The last time that the configuration of this row's endpoint was changed.
- *Types*: The number of types defined in this endpoint's schema.
- *Properties*: The number of properties defined in this endpoint's schema.
- *Actions*: Additional actions that you can take with this row's endpoint:
  - *Export schema definition*: Save a YAML schema definition file that can be imported into another endpoint configuration with the *Import schema definition file* button mentioned above.
  - *Configure endpoint*: Display configurable parameters for this endpoint. See *Endpoint configuration settings* below for details.

– *Delete endpoint*: Remove this endpoint from the repository's collection.

3.7.3.1 Endpoint configuration settings

When you select *Configure endpoint* from the *Actions* ··· menu on an endpoint's *Endpoint Management* row, this displays the *Configure GraphQL endpoint* dialog box. This dialog box includes an *Advanced settings* toggle that displays additional settings; the following shows a complete list of *Configure GraphQL endpoint* settings grouped in the categories used there:

GraphQL settings

- *Enable Type Count Queries*: Boolean property that controls whether count queries should be generated for all queryable types in the GraphQL schema. The queries are in the form of `<type>_count : Int` with support for filters. For example, if this is on, an `Employee` type in the schema would include an `employee_count` field that let you count the instances of that type with various options such as `distinct:` and `includeInferred:`. Disabled by default.
- *Enable Collection Count*: Boolean property that controls whether the multi-valued properties should have a companion property that provides counting of the collection properties. The new attributes are in the form of `<relation>_count : Int` with support for filters. Disabled by default.
- *Compact Error Messages*: Boolean property that enables more concise messages for certain types of errors. For example, some errors may cause messages for each object of a given type that causes the error; if this parameter is on, only one error message will be returned for each type that caused the error.
- *Default Fetch Language*: Default language spec (for example, `en`, `en,fr`, `~`,`NONE`, or `ANY`) to apply when loading values for the name property. If this is not set, all values will be returned. See *Using language specifications in your GraphQL queries* for examples of language specs.
- *Default Language Validation*: Set the default language validation rules to use for mutations of langString or stringOrLangString properties, including language uniqueness.
- *Default Implicit Language*: Determines the implicit language to use when inserting language-tagged strings if none is specified in a mutation.
- *Include Inferred Data*: Whether inferred data in the repository should be available via the endpoint.
- *Expand owl:sameAs*: Whether the endpoint should provide data based on triples inferred from `owl:sameAs` relationships.
- *Default Name Fetch Language*: Default language spec to apply when loading values for the GraphQL `name` property. The first value matching the specified language in order will be used. See *Using language specifications in your GraphQL queries* for examples of language specs.
- *Append Default Name Fetch*: When a GraphQL `name` property is mapped to an RDF property of type langString, which sometimes has multiple values for different languages, this helps to pick one. When this is set to on, the *Default Name Fetch Language* spec, if any, will be applied automatically for the name properties. See *Using language specifications in your GraphQL queries* for examples of language specs.

Security settings

- *Enable Mutations*: Whether mutations of data are allowed via the endpoint.
- *Default User Role*: Which RBAC role to use as a default.
- *Enable GraphQL Explain*: Whether to enable requests for *explain plans* to be generated by the underlying SPARQL query and returned with the GraphQL query results. Add these requests by including `(from: "onto:explain")` after an object name in a query, like this:

```
query MyQuery {
  employee(from: "onto:explain") {
  familyName
  }
}
```

Using `(from: "onto:sparql")` instead of `(from: "onto:explain")` will retrieve the generated SPARQL query instead of the explain plan.

- *Expose Model in GraphQL*: Includes model configuration details in the GraphQL schema. These will show up as additional information about the schema in GraphQL introspection queries.

Integration settings

- *SPARQL Federated Services*: Allows configuration of SPARQL federated service endpoints for the GraphQL endpoint. Enter this in JSON format with each service name being a key and the service address being the key's value.
- *SPARQL Federated Services Priority*: Determines the priority of SPARQL federated service endpoint configurations. Controls whether deployment-wide settings or individual endpoint configurations take precedence.
- *Query Prefix*: A prefix to be added to all query names in the GraphQL schema. For example, if this is set to "q1", a query that might otherwise have been generated as `query empQuery {employee {id}}` will be generated as `query q1_empQuery {employee {id}}`.
- *Mutation Prefix*: Similar to *Query Prefix* but adds prefixes to mutations.

3.7.4 Using language specifications in your GraphQL queries

Several GraphQL schema configuration parameters specify behavior using language specifications. These may be as simple as a single string naming a language like "fr" or they may be more complicated patterns. This section shows some of the possibilities.

Below are some examples of language specifications that return **one** value or no values. "First" means the first value that matches a condition. If there are several, one is picked at random.

Language Spec	Returns one value
`ANY`	first value with any language (or no language)
`~,NONE`	first value with non-empty language, then no language
`en`	first English-language value
`en,fr`	first English-language value, then a French-language value
`en-CA,en~,fr~`	first en-CA, then any English "dialect", then any French "dialect"
`en-CA,en~,~`	first en-CA, then any English "dialect", then any non-empty language
`en-CA,en~,ANY`	first en-CA, then any English, then any value
`en-CA,en~,~,NONE`	first en-CA, then any English, then any non-empty language, then empty language
`en,NONE`	first value in English, then a pure string (no language tag)
`NONE`	first pure string (no language tag)
`NONE,-~`	same, but more verbose
`~`	first value that has a non-empty language
`-NONE`	same
`~,-NONE`	same, but more verbose
`-fr`	first value that is not in French
`-fr~`	first value that is not in any French dialect
`BROWSER,NONE`	first value based on the user preferences or empty language

Examples of language specs that return **many** values. The language order is irrelevant.

Language Spec	Returns many values
(none)	return all values
`null`	same - return all values
`ALL`	same, but more verbose
`ALL:ANY`	same, but even more verbose
`ALL:en`	all English-language values
`ALL:en~`	all English-language or English "dialect" values
`ALL:~`	all values with language tag
`ALL:NONE`	all values without language tag (pure strings)
`ALL:en~,-en-US`	all English-language or English "dialect" values except American English
`ALL:BROWSER,en~`	all user preferred languages or English-language or English "dialect"

Note: When the `BROWSER` keyword is used with `ALL`, the language weights that come with the `Accept-Language` HTTP header are ignored and the order of the results is undetermined.

Examples of invalid language specs:

Language Spec	Error
`-ANY`	You cannot use language exclusion `-ANY` because that forbids all values
`ALL:ANY,en`	Language `en` is subsumed by `ANY` so it has no effect in `ALL` matching mode
`ALL:en,en~`	Language `en` is subsumed by `en~` so it has no effect in `ALL` matching mode
`en,ALL,fr`	Language mode `ALL` is allowed only in first position
`en~,en-GB`	Language `en-GB` is more specific than `en~` but comes later in the list so it has no effect
`en~,en-GB~`	Language `en-GB~` is more specific than `en~` but comes later in the list so it has no effect
`en,fr,bg,en`	Language `en` is specified more than once
`en-GB,en,-en~`	• Language exclusion `-en~` is more general than language `en-GB` so the language has no effect • Language exclusion `-en~` is more general than language `en` so the language has no effect

3.7.5 Using and managing GraphQL schemas with the GraphDB REST API

The GraphDB API section of the GraphDB API Swagger documentation lists two categories of API calls for working with the GraphQL API:

- The *GraphQL API* section shows calls that can list a repository's current GraphQL endpoints and API calls that send (via both `GET` and `POST` HTTP requests) GraphQL queries to default or non-default endpoints.
- The *GraphQL management* section shows REST API calls to manage the GraphQL models – for example, to generate, retrieve, edit, and report on them.

As an example, the following `curl` command uses the "List all active GraphQL endpoints" API call to retrieve information about the two GraphQL endpoints defined for the emp9c repository:

```
curl http://localhost:7200/rest/repositories/emp9c/graphql/endpoints
```

The result shows two endpoints in that repository, named EmpSchema" and "inventory":

```
[
        {
                "id": "EmpSchema",
                "active": true,
                "label": "Employee GraphQL Schema",
                "description": null,
                "lastModified": "2025-03-18T17:39:10.156Z",
                "objectsCount": 3,
                "propertiesCount": 7,
                "warnings": 0,
                "errors": 0,
                "default": true
        },
        {
                "id": "inventory",
                "active": true,
                "label": null,
                "description": null,
                "lastModified": "2025-03-18T19:24:25.921Z",
                "objectsCount": 0,
                "propertiesCount": 0,
                "warnings": 1,
                "errors": 0,
                "default": false
        }
]
```

The next `curl` command does an HTTP `POST` of the "Query specific GraphQL schema" API call to send an introspection GraphQL query for the names of all the types offered by the GraphQL API:

```
curl -X POST -H "Content-Type: application/json" \
  --data '{ "query": "query { __schema {  types {  name  } } } " }' \
  http://localhost:7200/rest/repositories/emp9c/graphql/EmpSchema
```

The result will be returned as a JSON block.

The GraphDB API Swagger documentation describes further details about the API calls that you can make — for example, parameters to pass and the structure of the expected results.

3.7.6 Role-Based Access Control

Having a GraphQL schema, you can control which information is returned or available for modification by a client's query using the built-in **Role-Based Access Control** (RBAC). Using RBAC, you can segregate duties within a team and grant the users the precise level of access necessary to perform their work.

The RBAC uses role definitions in the GraphQL schema to define access on a granular level. The role definitions map to *Custom user roles* in GraphDB. Whenever a client invokes a GraphQL query or mutation, the RBAC mechanism checks the user's custom roles against the rules defined for those roles in the schema.

RBAC generation is enabled by default. It is controlled by configuration `so:readRBACRules`.

Warning:

- The Role-Based Access Control requires a GraphQL license. In case the schema has defined RBAC, but the necessary license is missing, no operations will be executed.
- RBAC is applied only to users who are not Admins or Repo managers.

3.7.6.1 Role Definitions (Objects and Schema)

A role definition is a collection of triplets called actions with the format `<object>/<property>/<operation>`. It lists the object or property that can be read, written (created or updated), or deleted. Roles can be defined as high-level, such as owner or admin, or specific, such as a user with access to a particular property of a particular object.

Role definitions are declared in the GraphQL Schema. RBAC controls across the query and mutation of the objects or properties contained within a particular schema.

If the user does not have an assigned role, a default one will be assigned. Requests with missing role claims (where no role has been included) or roles that are not defined within the RBAC declaration will fall back to the Default role, which will allow all access.

Note: The behavior of the Default role can be redefined. To redefine the Default role, simply add it to the schema:

```
rbac:
  roles:
    Default:
       description: "Overwriting the Default role, which will allow read access to all users without a
↪role"
       notActions: [
         "*/*/read"
       ]
```

Actions and notActions are explained in further detail below.

3.7.6.2 Actions and notActions

Action permissions specify the data operations that the role is allowed to perform.

- If a role has **read** access to an operation with an object, then users with the corresponding custom role will be able to perform root level query, and query or read these object types within nested queries.
- If a role has **write** access to an operation with an object, then users with the corresponding custom role will be able to perform root level create or update mutation operation, and update these object types within nested queries.
- If a role has **write** access to an operation with an object, then users with the corresponding custom role will be able to perform root level delete mutation operation and delete these object types within nested queries.

The action grammar for is as follows:

```
{Object Regular Expression}/{Property Regular Expression}/read|write|delete|*
```

Currently, only `*` is allowed for `{Object Regular Expression}` and `{Property Regular Expression}`. Below you can find some examples of the action grammar:

Actions	Description
`"*/*/*"`	Read, write and delete access to all objects and all properties.
`"*/*/read"`	Read (Query) access to all objects and all properties.
`"*/*/write"`	Write (Create) access to all objects and all properties.
`"*/*/delete"`	Delete mutation operation access to all object and all properties.
`"Human/*/write"`	`Write access to all Human object properties`
`"Dro*/name*/read"`	Read access to all properties named `name*` for objects named `Dro*`.

While **actions** are used to allow access to an operation, **notActions** are used to specify the data operations that are excluded from the allowed actions. The access granted by a role (effective permissions) is computed by subtracting the `notActions` from the `actions` operations.

> **Warning:** `notActions` take precedence over `actions`.

The notAction grammar is otherwise the same as the one for actions:

```
{Object Regular Expression}/{Property Regular Expression}/read|write|delete
```

notActions defined for an abstract class are overridden by the child class.

For example, the following role definition would allow `DroidMgr` to perform any type of data operations with all Droid properties, with one exception — the `DroidMgr` roles would not be allowed to read the `Droid.name` property:

```
DroidMgr:
  description: "Role that accesses every property of Droid except name"
  actions: [
    "Droid/*/*"
  ]
  notActions: [
    "Droid/name/read"
  ]
```

3.7.6.3 RBAC Filters

If you need to fine-tune the role access, you can use filters with the RBAC actions. To add a filter, simply add a fourth section to the action containing the filter you wish to apply:

```
{Object Regular Expression}/{Property Regular Expression}/read|write|delete|*/(where:{Filter expression}
↪)
```

Filters can be used in all types of actions — read, write, and delete — as well as in mutations. Filters can also be combined — `action` filters are combined using `OR`, while `notAction` filters are combined using `AND`.

For example, let's take a look at the following rule:

```
HumanFromTatooineAdmin:
  description: "Can do everything with Tatooine and its residents."
    actions: [
      'Human/*/*/(where:{homeworld:{name:{EQ: "Tatooine"}}})',
      'Planet/*/*/(where:{name: {EQ: "Tatooine"}})'
    ]
```

It gives the following permissions:

- Delete `human` objects from `Tatooine`.
- Update `human` objects from `Tatooine`.
- Create `human` objects from `Tatooine`.
- Create, update or delete `Tatooine` planet itself.

Thus the following mutations are valid:

Delete and **Update**, since `https://swapi.co/resource/human/4` (Darth Vader) is from `Tatooine`:

```
mutation {
  delete_Human (where: {ID: "https://swapi.co/resource/human/4"}) {
    human {
      id
    }
  }
}
```

```
mutation {
  update_Human (
    objects: {
      desc: "From tatooine"
    },
    where: {ID: "https://swapi.co/resource/human/4"}) {
    human {
      id
    }
  }
}
```

However, you would not be allowed to update `human` objects that are not from `Tatooine`. So the following mutation will produce an error:

```
mutation {
  update_Human(
    objects: {
      desc: "Person from Tatooine"
    }
    where: {id: {}}
  ) {
    human {
      id
    }
  }
}
```

```
{
  "message": "ERROR: The UPDATE has been prevented as the selected objects of type 'Human' failed to
↪match the constraints '(where: [{homeworld: [{name: [{EQ:    \"Tatooine\"}]}]}])'",
  "locations": [
    {
      "line": 3,
      "column": 17
    }
  ]
}
```

The reason is that the applied filter (`where: {id: {}}`) will effectively select all `human` objects from the database, and not all of them are from `Tatooine`. So instead of updating only the `human` properties for which you have access, the whole mutation gets blocked and nothing gets updated.

3.8 Semantic similarity searches

3.8.1 What can the similarity plugin do for me?

The similarity plugin enables exploration and searching based on semantic similarity of RDF resources.

Statistical semantics queries can be valuable for:

- Taking text that is encoded as a literal in the database and finding the closest texts based on a vector space model. (The use of the vector space model is explained more below in *How does the similarity plugin work?*)
- Clustering stories from a news feed based on the events that they cover.

3.8.2 What does the similarity plugin do?

Humans determine the similarity between texts based on the similarity of the content words and their abstract meaning. Documents containing similar words are semantically related, and words frequently co-occurring are also considered close.

The similarity plugin supports document and term searches. A document is a literal or an aggregation of multiple literals, and a term is a word from a document.

There are four types of similarity searches:

- **Term to term** — supply a term to the plugin and it returns the closest semantically related terms
- **Term to document** — supply a term to the plugin and it returns the most representative documents for that term
- **Document to term** — supply a document to the plugin and it returns the most representative terms for that document
- **Document to document** — supply a document to the plugin and it returns a list of the closest related texts

3.8.3 How does the similarity plugin work?

The similarity plugin integrates the semantic vectors library and the underlying Random Indexing algorithm. The algorithm uses a tokenizer to translate documents to sequences of words (terms) and a vector space model to represent their abstract meaning. A distinctive feature of the algorithm is the dimensionality reduction approach based on Random Projection, where the initial vector state is generated randomly. With the indexing of each document, the term vectors are adjusted based on the contextual words. This approach makes the algorithm highly scalable for very large text corpora of documents, and research papers have proven that its efficiency is comparable to more sound dimensionality reduction algorithms like singular value decomposition.

After loading some news article data and creating an index, we're going to look at several kinds of searches that the similarity plugin can do.

3.8.4 Download sample data

The following examples use factforge.net news data and metadata from January to April 2018. These are represented as `content`, `creationDate`, and `mentionsEntity` triples.

1. Go to the SPARQL editor at http://factforge.net/sparql and run the following query to pull out the news story data. As you can see from the query's `CONSTRUCT` clause, for each news story the resulting dataset will have one or more mentioned entities, the news story content, and the article's creation date:

```
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ff-map: <http://factforge.net/ff2016-mapping/>

CONSTRUCT {
        ?document ff-map:mentionsEntity ?entity .
        ?document pubo:content ?content .
        ?document pubo:creationDate ?date .
} WHERE {
        ?document a pubo:Document .
        ?document ff-map:mentionsEntity ?entity .
        ?document pubo:content ?content .
        ?document pubo:creationDate ?date .
    FILTER (?p NOT IN (pubo:containsMention, pubo:hasFeature, pubo:hasImage))
```

(continues on next page)

(continued from previous page)

```
    FILTER ( (?date > "2018-01-01"^^xsd:dateTime) && (?date < "2018-04-30"^^
↪xsd:dateTime))
}
```

2. Use the *Download As* button in the lower right of the FactForge SPARQL editor to pull down a *Turtle* version of the query result. It may take a few minutes to export the data to the `query-result.ttl` file.
3. Open your GraphDB instance and *create a new repository* called "news".
4. Move the downloaded file to the `<HOME>/graphdb-import` folder so that it is visible in *Import* ▸ *Server files* (see how to *import server files*).
5. Import the `query-result.ttl` file into the "news" repository.
6. Go to *Setup* and enable the *Autocomplete index* for the "news" repository. It is used for autocompletion of URLs in the SPARQL editor and the *View resource* page.

3.8.5 Text-based similarity searches

3.8.5.1 Create text similarity index

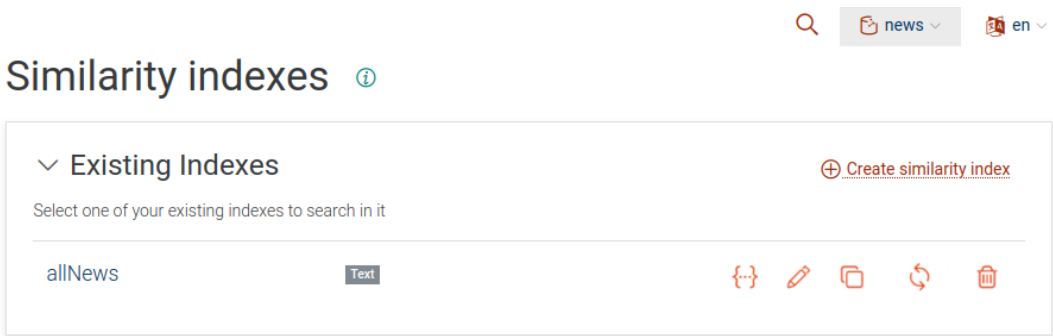
Create an index with the following steps:

1. Go to *Explore* ▸ *Similarity* ▸ *Create similarity index* ▸ *Create text similarity index* and change the *Data query* to:

```
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?documentID ?documentText
{
    ?documentID pubo:content ?documentText .
    FILTER(isLiteral(?documentText))
}
```

   This will index the content, where the ID of a document in the input data is the news piece's IRI and the text is the news piece's content. The `FILTER` condition shows that we only want triple objects that are literal values.

2. Name the index *allNews* and click the *Create* button in the lower right. An image of spinning arrows to the right of the new index's entry on the *Existing Indexes* list will show that the index is being created and not ready yet.
3. Once the index has been created, it will appear as a new choice on the the *Existing Indexes* list. This is where you select an index to use when searching.



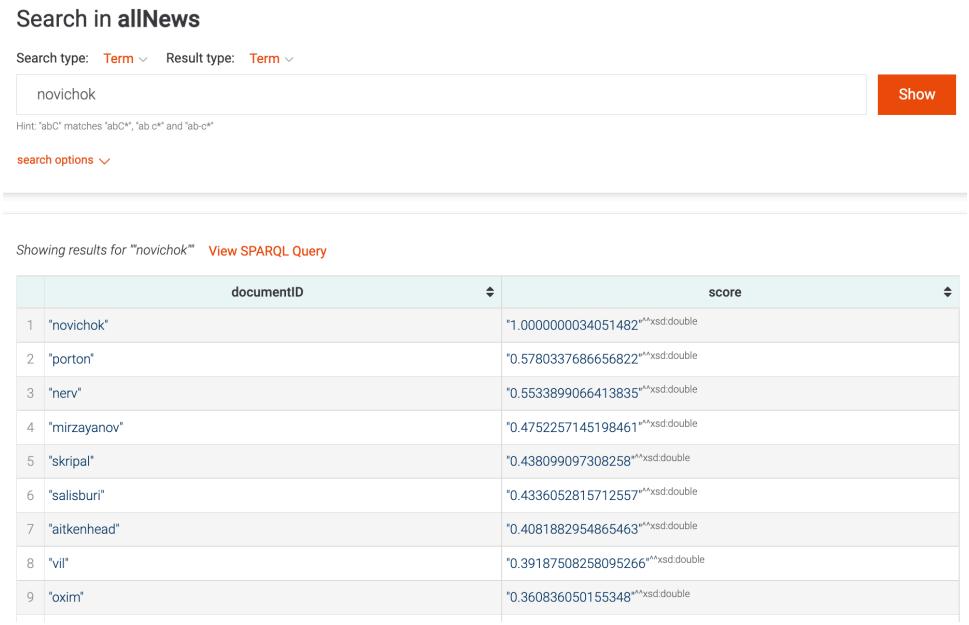
For each listed index, you can also do the following with the icons to the right of its name:

- {···} review or copy the SPARQL query that this index was created with
- modify the search query without having to build an index
- create a new index from an existing one
- rebuild the index
- delete the index

As described above, you can have the index search for a term and then return similar terms or documents, and you can also search for documents and have it return similar terms or documents. Select your new *allNews* index from the *Existing Indexes* list to try the examples below. This will show *Search type* and *Result type* drop-downs that each let you pick "Term" or "Document".

3.8.5.2 Use the index to search for similar terms

This example searches on the term "novichok" and requests similar terms. Returned results are sorted by their similarity score, which is shown in the second column of the results:

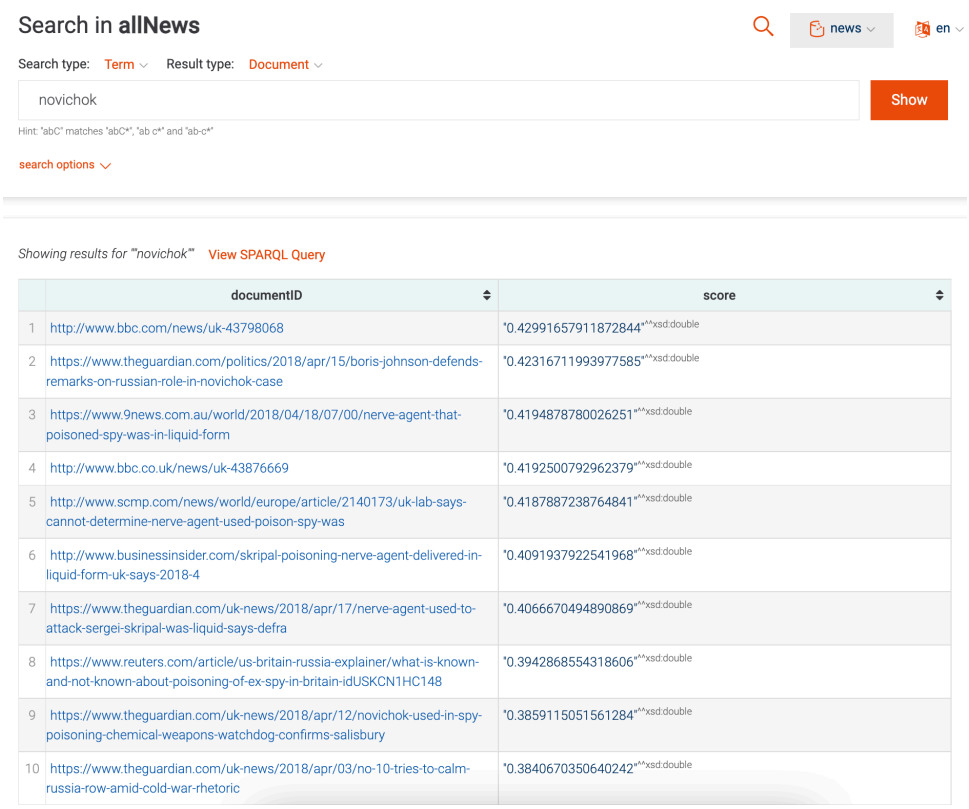


	documentID	score
1	"novichok"	"1.0000000034051482"^^xsd:double
2	"porton"	"0.5780337686656822"^^xsd:double
3	"nerv"	"0.5533899066413835"^^xsd:double
4	"mirzayanov"	"0.4752257145198461"^^xsd:double
5	"skripal"	"0.438099097308258"^^xsd:double
6	"salisburi"	"0.4336052815712557"^^xsd:double
7	"aitkenhead"	"0.4081882954865463"^^xsd:double
8	"vil"	"0.39187508258095266"^^xsd:double
9	"oxim"	"0.3608360501553 48"^^xsd:double

Note: For all searches done with the Similarity Index search form, the *View SPARQL Query* link above the search results shows you the SPARQL query generated by the form to create the displayed results. You can use this query and others based on it to incorporate these searches into your applications without relying on this form to do similarity searching.

3.8.5.3 Search documents for which selected term is specific

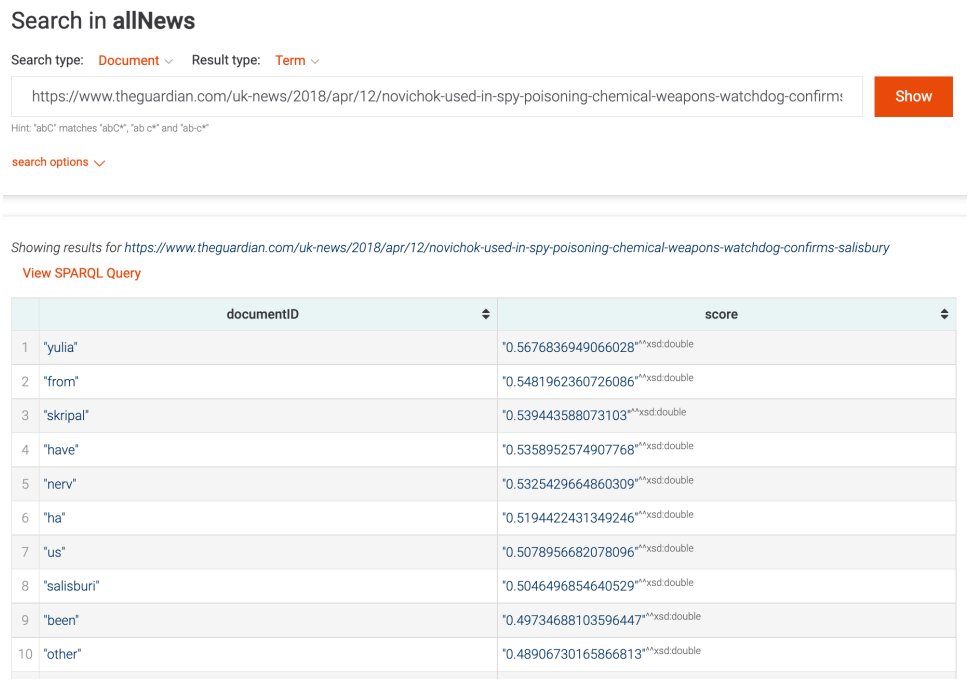
This example also searches for the term "novichok" but requests documents that are similar to that term. The result shows URLs of relevant documents and their similarity scores.



	documentID	score
1	http://www.bbc.com/news/uk-43798068	"0.42991657911872844"^^xsd:double
2	https://www.theguardian.com/politics/2018/apr/15/boris-johnson-defends-remarks-on-russian-role-in-novichok-case	"0.42316711993977585"^^xsd:double
3	https://www.9news.com.au/world/2018/04/18/07/00/nerve-agent-that-poisoned-spy-was-in-liquid-form	"0.4194878780026251"^^xsd:double
4	http://www.bbc.co.uk/news/uk-43876669	"0.4192500792962379"^^xsd:double
5	http://www.scmp.com/news/world/europe/article/2140173/uk-lab-says-cannot-determine-nerve-agent-used-poison-spy-was	"0.4187887238764841"^^xsd:double
6	http://www.businessinsider.com/skripal-poisoning-nerve-agent-delivered-in-liquid-form-uk-says-2018-4	"0.4091937922541968"^^xsd:double
7	https://www.theguardian.com/uk-news/2018/apr/17/nerve-agent-used-to-attack-sergei-skripal-was-liquid-says-defra	"0.4066670494890869"^^xsd:double
8	https://www.reuters.com/article/us-britain-russia-explainer/what-is-known-and-not-known-about-poisoning-of-ex-spy-in-britain-idUSKCN1HC148	"0.3942868554318606"^^xsd:double
9	https://www.theguardian.com/uk-news/2018/apr/12/novichok-used-in-spy-poisoning-chemical-weapons-watchdog-confirms-salisbury	"0.3859115051561284"^^xsd:double
10	https://www.theguardian.com/uk-news/2018/apr/03/no-10-tries-to-calm-russia-row-amid-cold-war-rhetoric	"0.3840670350640242"^^xsd:double

3.8.5.4 Search specific terms in selected document

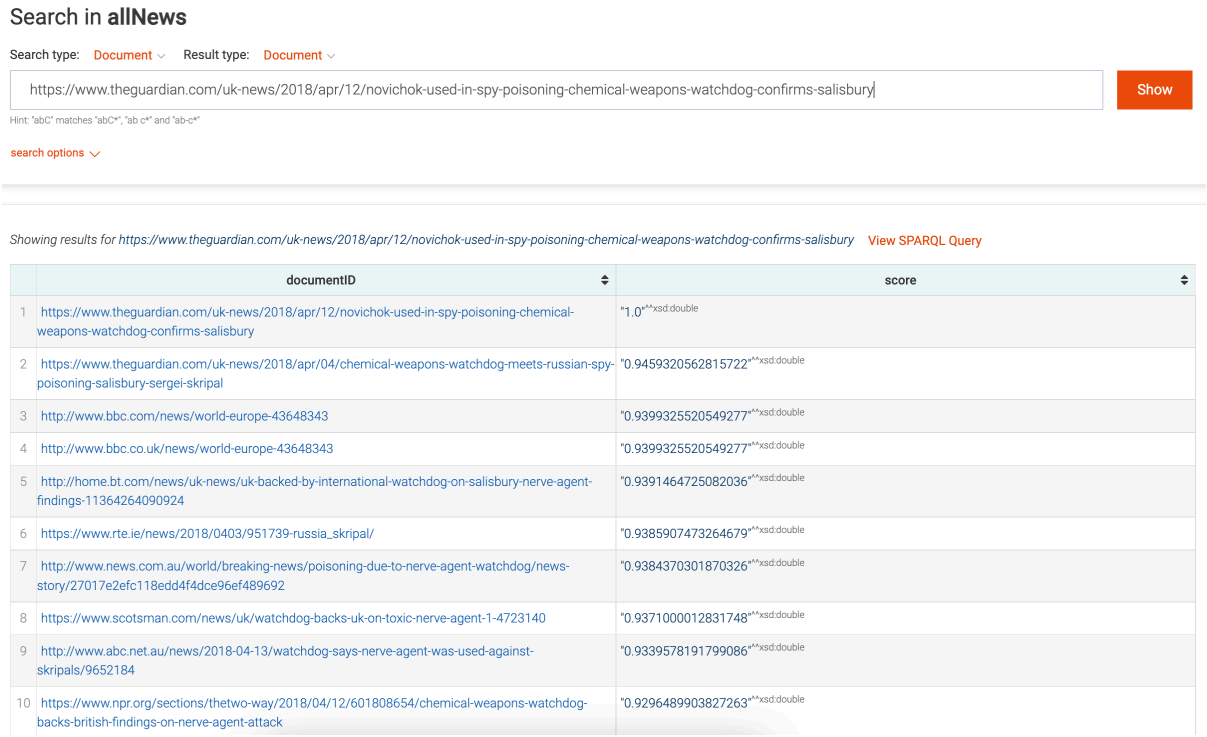
We saw that documents are identified by their URLs, and you could type out an entire URL to specify one as a *Search Type* value, but because you enabled autocomplete earlier you usually don't need to type out a full URL. With a *Search type* of *Document*, enter the words "used in spy" in the **allNews** index's search field, and autocomplete will enter the rest of the URL `https://www.theguardian.com/uk-news/2018/apr/12/novichok-used-in-spy-poisoning-chemical-weapons-watchdog-confirms-salisbury` for you. Click the *Show* button to find the most important words in the document and you'll see results similar to the following:



	documentID	score
1	"yulia"	"0.5676836949066028"^^xsd:double
2	"from"	"0.5481962360726086"^^xsd:double
3	"skripal"	"0.539443588073103"^^xsd:double
4	"have"	"0.5358952574907768"^^xsd:double
5	"nerv"	"0.5325429664860309"^^xsd:double
6	"ha"	"0.5194422431349246"^^xsd:double
7	"us"	"0.5078956682078096"^^xsd:double
8	"salisburi"	"0.5046496854640529"^^xsd:double
9	"been"	"0.4973468810359644 7"^^xsd:double
10	"other"	"0.4890673016586681 3"^^xsd:double

3.8.5.5 Search for closest documents

You can also search for documents that are closest to the specified search target document. Using the same input as the previous search, a search for the closest documents shows the results below. For result URLs that include title terms, we can see that many do have content related to the input document.



Note: Returned terms are sometimes stems of words such as "nerv" and "salisburi" in the example. This means that various words from the same word family may have been found.

3.8.5.6 Search similar news within a specified time range

We're going to execute a SPARQL query that searches for news stories similar to a particular one and then we're going to revise the query to only return stories published within two days of the input one.

1. For the initial search:
   a. In the similarity index list, click on the *allNews* index to search in it.
   b. Select *Search type: Document* and *Result type: Document*.
   c. In the search field, type `http://www.uawire.org/merkel-and-putin-discuss-syria-and-nord-stream-2` and click *Show*.
   d. Click the *View SPARQL Query* option above the returned results. It will contain the following query:

```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>

SELECT ?documentID ?score {
    ?search a inst:allNews ;
        :searchDocumentID <http://www.uawire.org/merkel-and-putin-discuss-syria-and-
↪nord-stream-2>;
        :searchParameters "";
        :documentResult ?result .
```

(continues on next page)

(continued from previous page)

```
    ?result :value ?documentID ;
            :score ?score.
}
```

e. Copy the query.

f. Paste it in Workbench's SPARQL editor.

2. Now, we can extend this search query to get only the news similar to `http://www.uawire.org/merkel-and-putin-discuss-syria-and-nord-stream-2` that have been created within days of the time of creation of this one, making it more likely to be the same news. Replace the query in the SPARQL editor with this augmented version of it and execute it:

```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?documentID ?score ?matchDate ?searchDate {
    BIND (<http://www.uawire.org/merkel-and-putin-discuss-syria-and-nord-stream-2> as ?
↪searchDocumentID )
    ?search a inst:allNews ;
        :searchDocumentID ?searchDocumentID;
        :searchParameters "";
        :documentResult ?result .
    ?result :value ?documentID ;
            :score ?score.
    ?documentID pubo:creationDate ?matchDate  .
        ?searchDocumentID pubo:creationDate ?searchDate .
    FILTER (?matchDate > ?searchDate - "P2D"^^xsd:duration && ?matchDate < ?searchDate
↪+ "P2D"^^xsd:duration)
}
```

The last three lines of the query do the following:

- Find the `pubo:creationDate` value of any documents found with the original search and store these values in the `?matchDate` variable.
- Store the `pubo:creationDate` value of the input document in `?searchDate`.
- Filter search results down to just those from up to two days before or two days after `?searchDate`.

We see the results below:

	documentID	score	matchDate	searchDate
1	http://www.uawire.org/merkel-and-putin-discuss-syria-and-nord-stream-2	"1.0"^^xsd:double	"2018-04-18T15:23:21Z"^^xsd:dateTime	"2018-04-18T15:23:21Z"^^xsd:dateTime
2	https://www.rferl.org/a/russia-germany-putin-merkel-discuss-syria-baltic-gas-pipeline/29172582.html	"0.921632613100701"^^xsd:double	"2018-04-17T12:00:00Z"^^xsd:dateTime	"2018-04-18T15:23:21Z"^^xsd:dateTime
3	https://carnegieeurope.eu/strategiceurope/76091	"0.8898680850620196"^^xsd:double	"2018-04-17T10:41:15Z"^^xsd:dateTime	"2018-04-18T15:23:21Z"^^xsd:dateTime

3.8.5.7 Term to term search

The *Term to term* search gets the relevant terms by period.

For this example we'll create four indexes — one each for the news in January, February, March, and April.

Go to *Create similarity index* and create a new index with the following query for January:

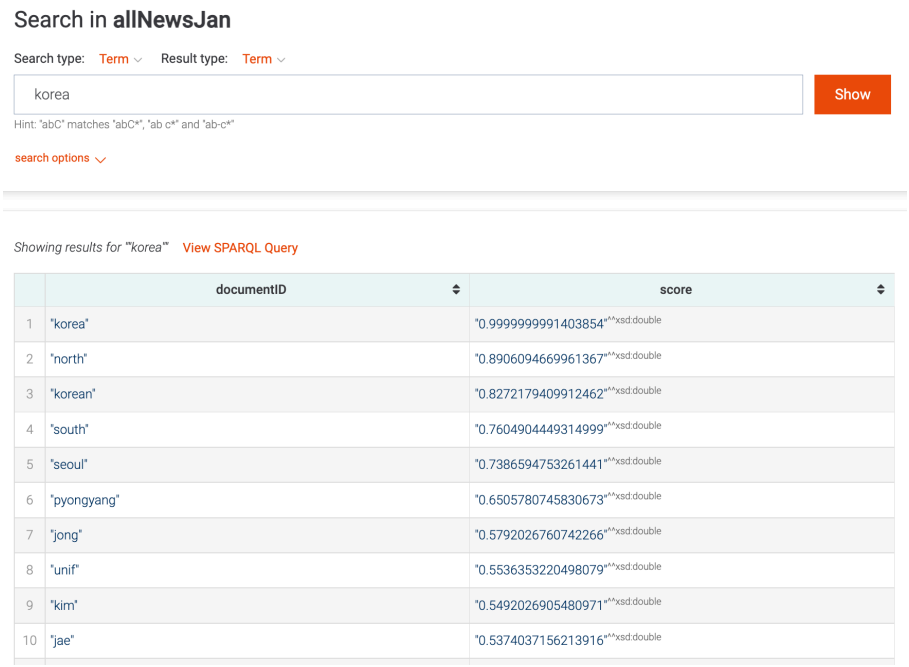
```
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
```

(continues on next page)

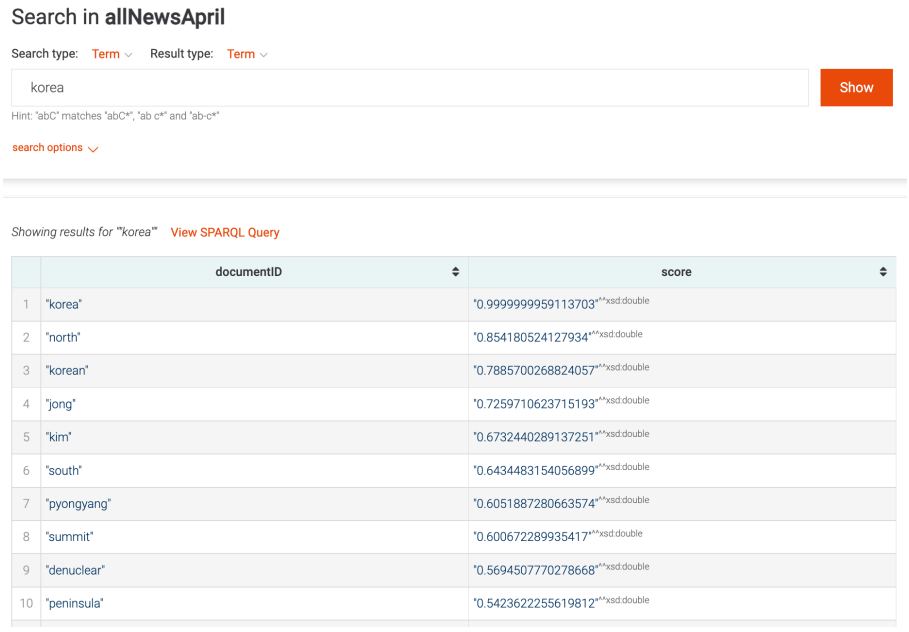
(continued from previous page)

```
SELECT ?documentID ?documentText {
    ?documentID pubo:content ?documentText .
    ?documentID pubo:creationDate ?date .
    FILTER ( (?date > "2018-01-01"^^xsd:dateTime) && (?date < "2018-01-30"^^xsd:dateTime))
    FILTER(isLiteral(?documentText))
}
```

Do the same for February, March, and April by changing the date range. After creating the four indexes, go to the index for each month and select *Term* for both *Search type* and *Result type*. Type "korea" in the search field. See how the results change over time.



	documentID	score
1	"korea"	"0.9999999991403854"^^xsd:double
2	"north"	"0.8906094669961367"^^xsd:double
3	"korean"	"0.8272179409912462"^^xsd:double
4	"south"	"0.7604904449314999"^^xsd:double
5	"seoul"	"0.7386594753261441"^^xsd:double
6	"pyongyang"	"0.6505780745830673"^^xsd:double
7	"jong"	"0.5792026760742266"^^xsd:double
8	"unif"	"0.5536353220498079"^^xsd:double
9	"kim"	"0.5492026905480971"^^xsd:double
10	"jae"	"0.5374037156213916"^^xsd:double

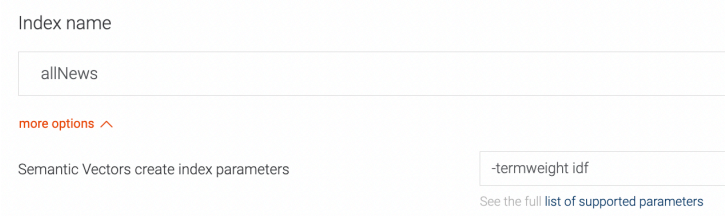


	documentID	score
1	"korea"	"0.9999999959113703"^^xsd:double
2	"north"	"0.854180524127934"^^xsd:double
3	"korean"	"0.7885700268824057"^^xsd:double
4	"jong"	"0.7259710623715193"^^xsd:double
5	"kim"	"0.6732440289137251"^^xsd:double
6	"south"	"0.6434483154056899"^^xsd:double
7	"pyongyang"	"0.605188728066357 4"^^xsd:double
8	"summit"	"0.6006722899354 17"^^xsd:double
9	"denuclear"	"0.5694507770278668"^^xsd:double
10	"peninsula"	"0.5423622255619812"^^xsd:double

3.8.6 Customizing index creation

3.8.6.1 Create index parameters

When you create a new index, you can select from a list of creation parameters under *more options* ▸ *Semantic Vectors create index parameters* to further configure the similarity index.



- **-vectortype**: Real, Complex, and Binary Semantic Vectors.
- **-dimension**: Dimension of semantic vector space, with a default value of 200. Recommended values are in the hundreds for real and complex, and in the thousands for binary, since binary dimensions are single bits. Smaller dimensions make both indexing and queries faster, but if the dimension is too low it will compromise the orthogonality of the element vectors and lead to poorer results. An intuition for the optimal values is given by the Johnson-Lindenstrauss lemma.
- **-seedlength**: Number of nonzero entries in a sparse random vector, with a default value 10 except for when vectortype is binary, in which case a default of `dimension/2` is enforced. For real and complex vectors the default value is 10, but it's a good idea to use a higher value when the vector dimension is higher than 200. The simplest thing to do is to preserve this ratio — that is, to divide the dimension by 20. (In the original implementation of random indexing, the ratio of non-zero elements was 1/3.)
- **-trainingcycles**: Number of training cycles used for Reflective Random Indexing.
- **-termweight**: Term weighting used when constructing document vectors. Values can be `none`, `idf`, `logentropy`, or `sqrt`. It's best to use term weighting when building indexes by adding `-termweight idf` as a default when creating an index. It uses inverse document frequency when building the vectors. See LuceneUtils for more details.
- **-minfrequency**: Minimum number of times that a term must occur in order to be indexed. The default value is 0, but it would be a bad idea to use it, because that could add a lot of big numbers, weird terms, and misspelled words to the list of word vectors. The best approach would be to set it as a fraction of the total word count in the corpus — for example 40 per million as a frequency threshold. Another approach is to start with an intuitive value, a single digit number like 3 or 4, and fine tune from there.
- **-maxfrequency**: Maximum number of times that a term can occur before getting removed from indexes. The default value is `Integer.MAX_VALUE`. Again, a better approach is to calculate it as a percentage of the total word count. Otherwise, you can use the default value and add most common English words to the stop list.
- **-maxnonalphabetchars**: Maximum number of non-alphabet characters a term can contain in order to be indexed. The default value is `Integer.MAX_VALUE`. Recommended values depend on the dataset and the type of terms it contains, but setting it to 0 works pretty well for most basic cases, as it takes care of punctuation (if data has not been preprocessed), malformed terms, and weird codes and abbreviations.
- **-filternumbers**: `true/false`, whether to index numbers.
- **-mintermlength**: Minimum number of characters in a term.
- **-indexfileformat**: Format used for serializing and deserializing vectors from disk. The default value is `lucene`. Another option is `text`, which may be used when debugging to see the actual vectors, although this can be too slow with real data.

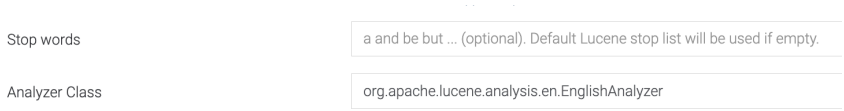
3.8.6.2 Disabled parameters

- **-luceneindexpath**: Currently, you are not allowed to build your own Lucene index and create vectors from it because the creation of the index and vectors is all done in one step.
- **-stoplistfile**: Replaced by the `<http://www.ontotext.com/graphdb/similarity/stopList>` predicate. Stop words are passed as a string literal as opposed to a file.
- **-elementalmethod**
- **-docindexing**

3.8.6.3 Stop words and Lucene analyzer

In the *Stop words* field, you can add a custom list of stop words to be passed to the Semantic Vector plugin. If left empty, the default Lucene stop words list will be used.

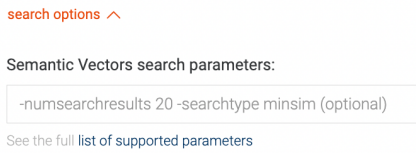
The *Analyzer class* field lets you can set a Lucene analyzer to use during Semantic Vector indexing and query time tokenization. The default is `org.apache.lucene.analysis.en.EnglishAnalyzer`, but it can be any from the supported list.



The Lucene connector also supports custom Analyzer implementations. This lets you create your own analyzer and add it to a classpath. The value of the *Analyzer Class* parameter must be a fully-qualified class name that extends the Java class `org.apache.lucene.analysis.Analyzer`.

3.8.6.4 Search parameters

On the search form, expanding the *Search options* panel lets you configure more search parameters.



- **-searchtype**: Most search types involve processing combinations of vectors in different ways, building a query expression, scoring candidates against these query expressions, or both. The default is a sum that builds a query by adding together weighted vectors for each of the query terms, and search using cosine similarity.
- **-matchcase**: If `true`, matching of query terms is case-sensitive; otherwise, case-insensitive. Default: `false`.
- **-numsearchresults**: Number of search results.
- **-searchresultsminscore**: Search results with similarity scores below this threshold will not be returned. Default: `-1`.

See more about Semantic Vectors Search Options.

3.8.6.5 Delete an index using a SPARQL query

To **delete** an index, use the following SPARQL query, substituting your index name for `my_index`:

```
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>

INSERT DATA {
    similarity-index:my_index similarity:deleteIndex "" .
}
```

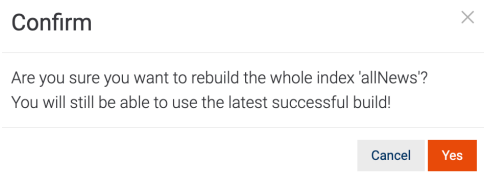
3.8.6.6 Rebuild an index using a SPARQL query

To **rebuild** an index, use the following SPARQL query, substituting your index name for `my_index`:

```
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>

INSERT DATA {
    similarity-index:my_index similarity:rebuildIndex "" .
}
```

3.8.6.7 Search in the index during rebuild with no downtime

GraphDB lets you use the similarity index with no downtime while the database is being modified. While rebuilding the index, its last successfully built version is preserved until the new index is ready. This way, when you search in it during rebuild, the retrieved results will be from this last version. The following message will notify you of this:



The outdated image is then replaced.

3.8.6.8 Locality-sensitive hashing

Note: As locality-sensitive hashing does not guarantee the retrieval of the most similar results, this hashing is not the most suitable option if precision is essential. Hashing with the same configuration over the same data does not guarantee the same search results.

Locality-sensitive hashing can reduce search times. Without a hashing algorithm, a search consists of the following steps:

1. A search vector is generated.
2. All vectors in the store are compared to this search vector, and the most similar ones are returned as matches.

While this approach is complete and accurate, it is also time-consuming. Hashing can speed the process by reducing the number of candidates for most similar vectors. This is where Locality-sensitive hashing can help.

The Locality-sensitive hashing algorithm has two parameters that can be passed either during index creation, or as search options:

- **-lsh_hashes_num**: The number of `n` random vectors used for hashing. Default: `0`

- **-lsh_max_bits_diff**: The `m` number of bits by which two hashes can differ and still be considered similar. Default: `0`

The hashing workflow is as follows:

1. An `n` number of random orthogonal vectors are generated.
2. Each vector in store is compared to each of those vectors (checking whether their scalar product is positive or not).
3. Given this data, a hash is generated for each of the vectors in store.

A search workflow follows these steps:

1. A search vector is generated.
2. A hash is generated for this search vector by comparing it to the `n` number of random vectors used during the initial hashing.
3. All hashes similar to that of the searched vector are found. (A hash is considered similar when it has up to `m` bits difference from the original one.)
4. All vectors with such hash values are collected and compared to the generated vector in order to get the closest ones, based on the assumption that the vectors with similar hashes will be close to each other.

Note: If both parameters have the same value, then all possible hashes are considered similar and therefore no filtering is done. For optimization purposes in this scenario, the entire hashing logic has been bypassed.

If one of the parameters is specified during the index creation, then its value will be used as the default one for searching.

Depending on its configuration, the hash can perform in different ways.

A higher **-lsh_hashes_num** value leads to more hash buckets with fewer elements in them. Conversely, a lower number of hashes would mean fewer but bigger buckets. The `n` number of hashes leads to `2^n` potential buckets.

A higher **-lsh_max_bits_diff** value leads to more buckets being checked and vice versa. More precisely, an `m` number of **-lsh_max_bits_diff** with an `n` number of hashes leads to m-combinations of `n + (m - 1)` -combination of `n + ... + 0` -combinations of `n` checked buckets.

By modifying these parameters, you can control the number of checked vectors. A lower number of checked vectors leads to higher performance, but also increases the chance of missing a similar vector.

Different settings perform well for different vector store sizes. A reasonable initial configuration is `(3, 1)`. If you want to slightly increase the precision, you can change it to `(3, 2)`, but this will substantially increase the number of checked vectors and reduce performance.

To make finer calibration, you would need a higher number of hashes — for instance, `(6, 2)` is also a possible configuration.

To increase performance, you could change the configuration to `(6, 1)` or `(8, 2)`, but this will reduce precision.

If increasing the precision at the cost of performance is an acceptable option for you, you could use the configuration of `(6, 3)`.

Note: If the **-lsh_max_bits_diff** value is too close to the **-lsh_hashes_num** value, the performance can suffer compared to the default one because of the computational overhead.

3.8.6.9 Boosting a term's weight

You can boost a given term's weight in the text-based similarity index for term-based searches (*Term to term* or *Term to document*). Boosting a term's weight can be done by using the caret symbol `^` followed by a boosting factor — a positive decimal number `term^factor`.

For example, `UK Brexit^3 EU` will perform a search in which the term "Brexit" will have 3 times more weight than "UK" and "EU", and you can expect the results to be mainly related to "Brexit".

The default boosting factor is `1`. Setting a boosting factor of `0` will completely ignore the given term. Escaping the caret symbol `^` is done with a double backslash `\\^`.

Note: The boosting will not work in document-based searches (*Document to term* or *Document to document*), meaning that the caret following by a number will not be treated as a weight boosting symbol.

3.8.7 Predication-based Semantic Indexing

Predication-based Semantic Indexing, or PSI, is an application of distributional semantic techniques for reasoning and inference. PSI starts with a collection of known facts or observations and combines them into a single semantic vector model in which both concepts and relationships are represented. With this technique, the usual ways for constructing query vectors and searching for results in semantic vectors can be used to suggest similar concepts based on the knowledge graph.

3.8.7.1 Load example data

The predication-based semantic search examples are based on *Person data* from the DBpedia dataset. The sample dataset contains over 730,000 triples for over 101,000 persons born between 1960 and 1970.

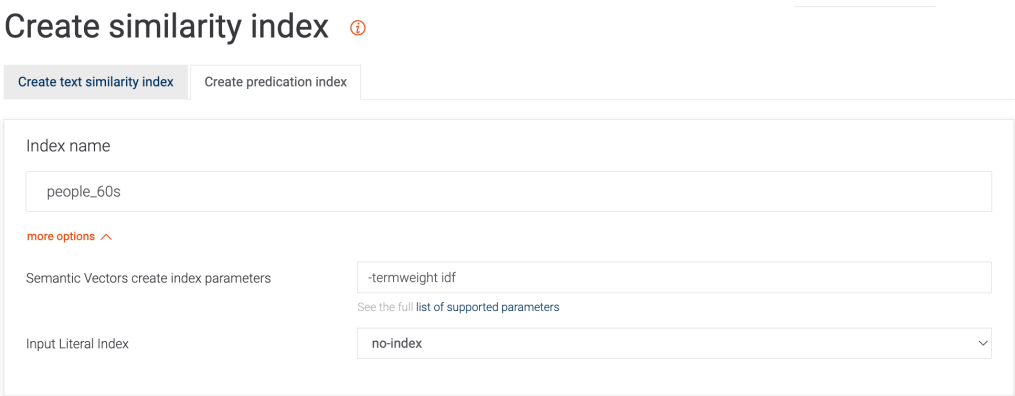
1. Download the provided `persons-1960-1970` dataset.
2. Unzip it and import the `.ttl` file into a repository.
3. Enable the Autocomplete index for the repository from *Setup* ▸ *Autocomplete*.

For ease of use, you may add the following namespaces for the example dataset (done from *Setup* ▸ *Namespaces*):

- dbo: `http://dbpedia.org/ontology/`
- dbr: `http://dbpedia.org/resource/`
- foaf: `http://xmlns.com/foaf/0.1/`

3.8.7.2 Create a predication-based index

1. From *Explore* ▸ *Similarity* ▸ *Create similarity index*, select *Create predication index*.



2. Fill in the index name shown above and under *more options* add `-termweight idf` in the *Semantic Vectors create index parameters* field so that the index uses term weighting. As another parameter option, setting `-dimension` higher than the default value of 200 in that same field can improve search results.

3. Configure the *Data query*. This `SPARQL SELECT` query determines the data that will be indexed. The `SELECT` query must bind the following variables:

   - `?subject`
   - `?predicate`
   - `?object`

   The *Data query* is executed during index creation to obtain the actual data for the index. When data in your repository changes, you must rebuild the index to have it reflect the changes. It is a subquery of a more complicated query that you can see with the *View Index Query* button.

   For the given example, leave the default *Data query*. This will create an index with all triples in the repository:

```
SELECT ?subject ?predicate ?object
  WHERE {
      ?subject ?predicate ?object .
  }
```

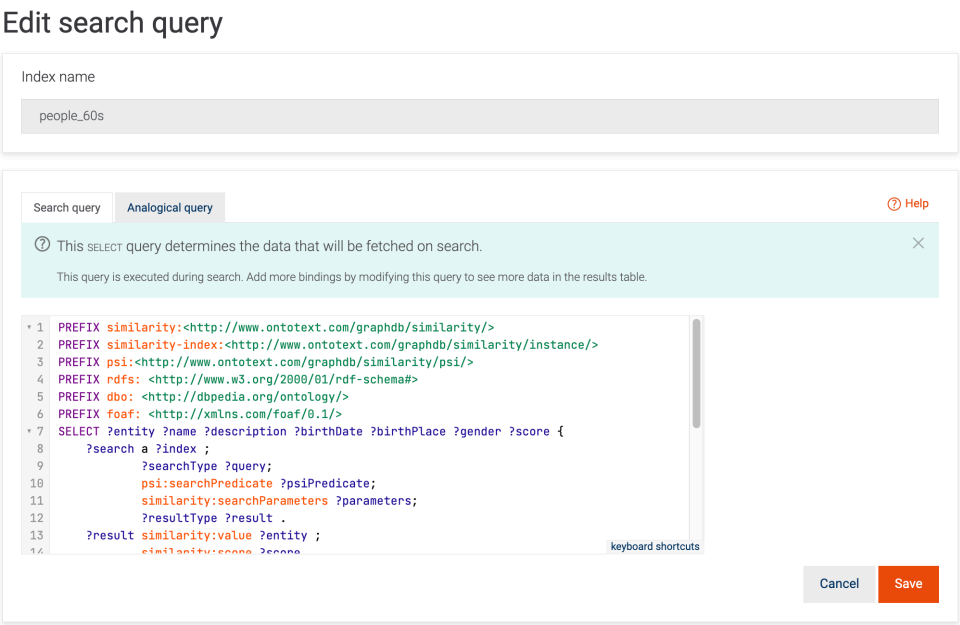
4. Set the *Search query*. This `SELECT` query determines the data that will be fetched on search. Add more bindings by modifying this query to see more data in the results table.

   For this example, set the *Search query* to:

```
PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX psi: <http://www.ontotext.com/graphdb/similarity/psi/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?entity ?name ?description ?birthDate ?birthPlace ?gender ?score {
    ?search a ?index ;
        ?searchType ?query;
        psi:searchPredicate ?psiPredicate;
        similarity:searchParameters ?parameters;
    ?resultType ?result .
    ?result similarity:value ?entity ;
        similarity:score ?score .
    ?entity foaf:name ?name .
    OPTIONAL { ?entity <http://purl.org/dc/terms/description> ?description . }
    OPTIONAL { ?entity dbo:birthPlace ?birthPlace . }
    OPTIONAL { ?entity dbo:birthDate ?birthDate . }
    OPTIONAL { ?entity foaf:gender ?gender . }
}
```

5. Click *Create* to start index creation.

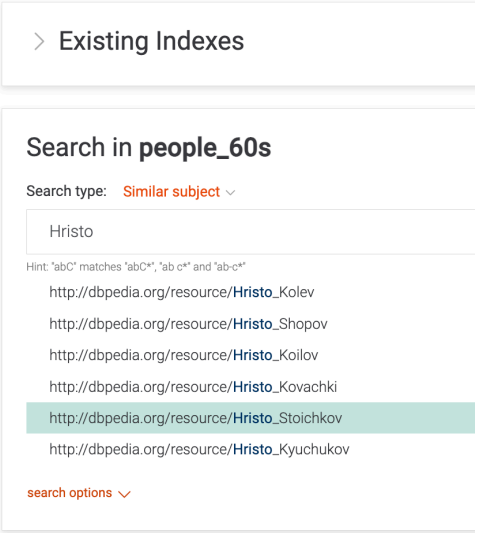
Once the index has been built, you have the same options as for the *text similarity index*: *View SPARQL query*, *Edit query*, *Create index from existing one*, *Rebuild*, and *Delete* index. If you want to edit an index query, you can do it for both the *Search* and the *Analogical* queries, which are described below.



3.8.7.3 Search predication-based index

In the list of *Existing indexes*, select the *people_60s* index that you will search in.

In our example, we will look for individuals similar to Hristo Stoichkov — the most famous Bulgarian football player.



In the results, you can see Bulgarian football players born in the same town, other Bulgarian athletes born in the same place, as well as other people with the same birth date.

Showing results for http://dbpedia.org/resource/Hristo_Stoichkov View SPARQL Query

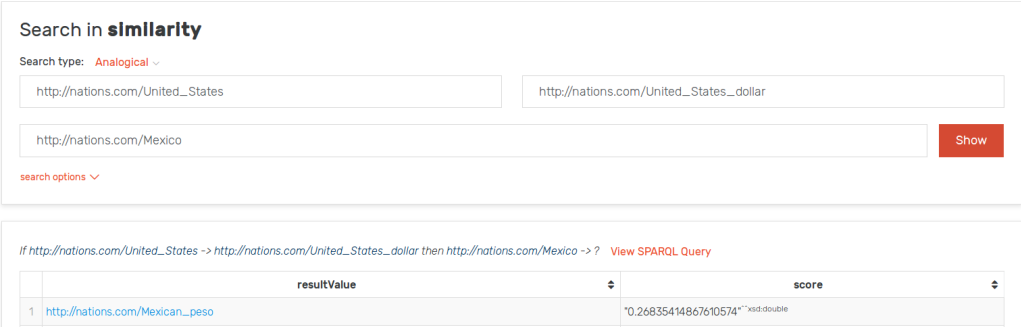
	entity	name	description	birthDate	birthPlace	gender	score
1	dbr:Hristo_Stoichkov	"Hristo Stoichkov"@en	"Bulgarian footballer"@en	"1966-02-08"^^xsd:date	dbr:Plovdiv	"male"@en	"1.0000000001402665"^^xsd:double
2	dbr:Yasen_Petrov	"Yasen Petrov"@en	"Bulgarian footballer"@en	"1968-06-23"^^xsd:date	dbr:Plovdiv	"male"@en	"0.5256592804986613"^^xsd:double
3	dbr:Hristo_Kolev	"Hristo Kolev"@en	"Bulgarian footballer"@en	"1964-09-21"^^xsd:date	dbr:Plovdiv	"male"@en	"0.4014514816274087"^^xsd:double
4	dbr:Hristo_Koilov	"Hristo Koilov"@en	"Bulgarian footballer"@en	"1969-09-23"^^xsd:date	dbr:Plovdiv	"male"@en	"0.39336816621445825"^^xsd:double
5	dbr:Kiril_Andonov	"Kiril Andonov"@en	"Bulgarian footballer"@en	"1967-11-01"^^xsd:date	dbr:Plovdiv	"male"@en	"0.39259702960672094"^^xsd:double

3.8.7.4 Analogical searches

Along with searching explicit relations and similarities, PSI can also be used for analogical search.

Suppose you have a dataset with currencies and countries, and want to know the following: "If I use dollars in the USA, what do I use in Mexico?" By using the predicate index, you do not need to know the predicate ("has currency").

1. Import the `Nations.ttl` sample dataset into a repository.
2. Build an Autocomplete index for the repository.
3. Build a predication index following the *steps above*. (There is no need to set the *Search query* because you'll be using an Analogical query instead.)
4. Once the index is built, you can use the *Analogical* search option of your index. In logical terms, your query will translate to "If USA implies dollars, what does Mexico imply?"



As you can see, the first result is peso, the Mexican currency. The rest of the results are not relevant in this situation since they are part of a very small dataset.

3.8.7.5 Why is this important?

PSI supplements traditional tools for artificial inference by giving "nearby" results. In cases where there is a single clear winner, this is essentially the behavior of giving "one right answer". But in cases where there are several possible plausible answers, having robust approximate answers can provide great benefit.

3.8.8 Hybrid indexing

A Predication index creates a random vector for each entity in the database and uses these vectors to generate the similarity vectors that will be used for similarity searches. This approach does not take into consideration the similarity between the literals themselves. Let's examine the following example, using the FactForge data from above:

```
<express:donald-tusk-eu-poland-leave-european-union-polexit> <pubo:formattedDate> 1/11/2018
<telegraph:donald-tusk-warnspoland-could-hold-brexit-style-eu-referendum> <pubo:formattedDate> 1/11/2018
<express:POLAND-s-bid-for-World-War-2-reparations-is-bolstered-by-a-poll-which-found-that-a-majorit>
↪<pubo:formattedDate> 1/6/2018
```

Naturally we would expect the first news article to be more similar to the second one than to the third one, not only based on their topics — Poland's relationship with the EU — but also because of their dates. However, the normal Predication index would not take into account the similarity of the dates, and all news would have fairly close scores. In order to handle this type of scenario, we can first create a Text similarity index. It will find that the dates of the three articles are similar, and will then use this information when building the Predication index.

The following describes how to create one.

3.8.8.1 Edit the FactForge data

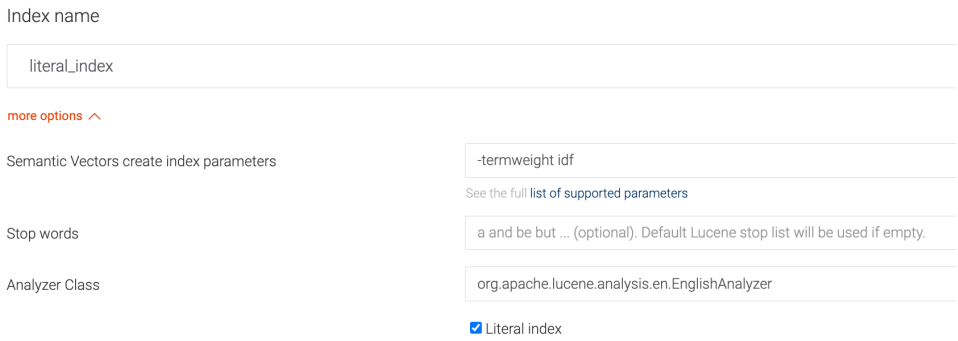
Dates, as presented in FactForge, are not literals that the similarity plugin can handle easily. This is why you need to format them to something easier to parse. Do this with the following query on the Workbench SPARQL view to insert `pubo:formattedDate` triples with a valueof the form `mm/dd/yyyy`:

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
INSERT {
    ?x pubo:formattedDate ?displayDate
}
WHERE {
    ?x pubo:creationDate ?date.
    BIND (CONCAT(STR(MONTH(?date)),
                 "/",
                 STR(DAY(?date)),
                 "/",
            STR(YEAR(?date))) as ?displayDate)
}
```

Go to *Setup* and enable the *Autocomplete index* for *allNews*.

3.8.8.2 Create a Literal index

The Literal index is a subtype of the Text index. To build it, create a normal Text index by ticking the *Literal index* checkbox from the *More options* menu. These types of indexes can only be used as input indexes for predication indexes and will be indicated in the Similarity page. They can not be used for similarity searching. The index will include all literals returned by the `?documentText` variable from the Data query.



Make sure to filter out the mentions, so the data in the Literal index only contains the news. When creating the index, use the following *Data query*:

```
SELECT ?documentID ?documentText {
        ?documentID ?p ?documentText .
        filter(isLiteral(?documentText))
        filter (?p != <http://factforge.net/ff2016-mapping/mentionsEntity>)
}
```

Click *Create* to create the index. It may take a few minutes, so make sure to wait until it finishes before proceeding.

3.8.8.3 Use the Literal index

Create a new predication index. From its *More options* menu, select *Input Literal Index* -> the index created in the previous step.

Since you do not want to look at mentions, and in this sense the default data format is useless, you need to filter them out from the data used in the predication index. Add the following Data query:

```
SELECT ?subject ?predicate ?object
WHERE {
    ?subject ?predicate ?object .
        filter (?predicate != <http://factforge.net/ff2016-mapping/mentionsEntity>)
        filter (?predicate != <http://ontology.ontotext.com/publishing#creationDate>)
}
```

For the purposes of the test, we want to also display the new formatted date when retrieving data. Go to the search query tab and add the following query:

```
PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
```

(continues on next page)

(continued from previous page)

```
PREFIX psi: <http://www.ontotext.com/graphdb/similarity/psi/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>

SELECT ?entity ?score ?content ?date {
    ?search a ?index ;
        ?searchType ?query;
        psi:searchPredicate ?psiPredicate;
        similarity:searchParameters ?parameters;
        ?resultType ?result .
    ?result similarity:value ?entity ;
            similarity:score ?score .
    ?entity pubo:content ?content .
    ?entity pubo:formattedDate ?date .
}
```

With those two queries in place, the data returned from the index should be more useful. Create your hybrid predication index and wait for the process to be completed. Then, open it and run a query for "donald tusk", selecting the Express article about "Polexit" from the Autocomplete suggest box. You will see that the first results are related to the Polexit and dated the same.

3.8.8.4 Indexing behavior

When building the Literal index, it's a good idea to index all literals that will be indexed in the Predication index, or at least all literals of the same type. Continuing with the example above, let's say that the Literal index you have created only returns these three news pieces. Add the following triple about a hypothetical Guardian article and create a Predication index to index all news:

```
<guardian:poland-grain-exports> <pubo:formattedDate> 12/08/2017
```

Based on the triples, one would expect that the first article will be equally similar to the third and the new one — their contents and dates have little in common. However, depending on the binding method used when creating the Predication index, you can get a higher score for the third article compared to the new one only because the third article has been indexed by the Literal index. There are two ways to easily avoid this: either index all literals, or at least index all the dates.

3.8.8.5 Manual creation

If you are not using the Similarity page, you could pass the following options when creating the indexes:

- `-literal_index true`: passed to a Text index creates a Literal index
- `-input_index <literaIndex>` (replacing `<literalIndex>` with the name of an existing Literal index): passed to a Predication index creates a hybrid index based on a Literal index

3.8.9 Using training cycles

When building Text and Predication indexes, you can use training cycles to increase the accuracy of the index. The number of training cycles can be set by passing the option `-trainingcycles <numOfCycles>`. The default number of training cycles is 0.

Text and Predication indexes have quite different implementations of the training cycles.

Text indexes just repeat the same algorithm multiple times, which leads to algorithm convergence.

Predication indexes initially start the training with a random vector for each entity in the database. On each cycle, the initially random elemental vectors are replaced with the product of the previous cycle, and the algorithm is run again. In addition to the entity vectors, the predicate vectors get trained as well. This leads to higher computational time for a cycle compared to the initial run with `trainingcycles = 0`.

Note: Each training cycle costs time and computing cycles, and a higher number of cycles will greatly increase the building time.

3.9 Querying GPT models

A Generative Pretrained Transformer (GPT) is a type of Large Language Model (LLM) that supports natural language querying of a model that is typically trained on a very large collection of data. This makes it possible for queries to address a wide range of topics.

GPT models, popularized by OpenAI, let users type in questions and see answers in response to those questions. GraphDB provides a set of *magic predicates*, implemented as extension functions, that let your SPARQL queries communicate with GPT models. This lets you combine the power of these models with your own knowledge graphs.

3.9.1 Prerequisites and additional configurations

You must obtain an appropriate API key and set the `graphdb.llm.api-key` value before you can use the GPT functions listed below in your SPARQL queries.

See also:

For more information on how to configure your copy of GraphDB and change the necessary properties:

- *LLM properties*
- *Talk to Your Graph properties*
- *SPARQL functions and SPARQL explain*

3.9.2 GPT functions

The GPT chat functions send your queries to the GPT model that you have set in your configuration, as described below, and return the answer as part of your SPARQL query results.

These functions are implemented as *magic predicates*. These look like triple patterns, but arguments are passed in the object position and the result is bound to the variable or parenthesized variable list in the subject position.

In the syntax examples below, the `gpt:` prefix represents the URI `http://www.ontotext.com/gpt/`.

3.9.2.1 Setting the temperature of the response

The last argument passed to `gpt:ask`, `gpt:list`, or `gpt:table` can be a real number between 0 and 2 that sets the temperature of the response. Higher values such as 1.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic. The default value is 1.

3.9.2.2 `gpt:ask()` — Retrieve a single answer

The `gpt:ask()` function passes one or more messages with instructions to the GPT model. The results (unlike with *gpt:list*) are stored in a single binding. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response.

Because `gpt:ask()` is a magic predicate, you call it as a triple pattern with the variable in which to store the response as the subject and the messages to pass (enclosed in parentheses if there are more than one) as the object:

```
?answer gpt:ask ?message
?answer gpt:ask (?message1 ?message2 ...)
```

The following SPARQL query passes two messages in the triple pattern's object position:

```
PREFIX gpt: <http://www.ontotext.com/gpt/>

SELECT * WHERE {
    ?primes gpt:ask ("List three prime numbers."
                     "Make them between 100 and 200.")
}
```

Note how the result of this query includes three numbers, but only one value is actually returned: a single quoted string with those numbers:

	primes
1	"The three prime numbers between 100 and 200 are 107, 109, and 113."

If the execution of your query binds a variable multiple times because of the use of another data source, GraphDB will call `gpt:ask` for each one. This provides an excellent way for a query to combine information from your own knowledge graph with a GPT model.

The following example lists the types of grape stored in a `wine dataset`. It then creates a query with each of those names to send to the GPT about what brand of wine uses that grape:

```
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX gpt:  <http://www.ontotext.com/gpt/>

SELECT ?brands WHERE {

    ?grape a wine:Grape ;
           rdfs:label ?grapeName .

    BIND(CONCAT("What brand of wine uses the ", ?grapeName, " grape?") AS ?query)
    ?brands gpt:ask ?query .
}
```

The following shows the first three rows of the result:

	brands
1	"The Merlot grape is used by many different wine brands around the world, including some of the most well-known and respected names in the industry. Some popular examples include Chateau Petrus, Chateau Lafite Rothschild, and Chateau Cheval Blanc from France, as well as many wineries in California, Washington state, and other regions around the world."
2	"Many brands of wine use the Cabernet Sauvignon grape. Some popular brands include Chateau Margaux, Chateau Lafite Rothschild, Opus One, Silver Oak, Kendall-Jackson, and Robert Mondavi."
3	"Many brands of wine use the Cabernet Franc grape, but some popular ones include Chateau Cheval Blanc, Chateau Angélus, and Chinon."

Queries may use punctuation for more structure, but there is no specific syntax to follow. All messages are treated as text. For example, the following query lists National Basketball Association players from Australia who have played in the guard position:

```
prefix gpt: <http://www.ontotext.com/gpt/>

SELECT * WHERE {
    ?player gpt:ask ("List NBA players. nationality: Australian. position: guard." )
}
```

3.9.2.3 `gpt:list()` — Retrieve a list of answers

The `gpt:list()` function passes one or more messages with instructions to the GPT model. These instructions should name a specific number of results that you would like. The results (unlike with *gpt:ask*) can then be returned as multiple bindings of the specified variable, which will be displayed as separate result set rows. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response.

Because `gpt:list()` is a magic predicate, you call it as a triple pattern with the variable in which to store the response as the subject and the messages to pass (enclosed in parentheses if there are more than one) as the object:

```
?answer gpt:list ?message
?answer gpt:list (?message1 ?message2 ...)
```

The following example SPARQL query is very similar to the first example shown in the description of the *gpt:ask()* magic predicate, but it calls `gpt:list()` instead:

```
PREFIX gpt: <http://www.ontotext.com/gpt/>

SELECT * WHERE {
    ?primes gpt:list ("List three prime numbers."
    "Make them between 100 and 200.")
}
```

Note how the result set has four rows, unlike the `gpt:ask()` version of the query — one for an introduction and one for each prime number returned:

	primes
1	"Here are three prime numbers between 100 and 200:"
2	"101"
3	"127"
4	"131"

Tip: The result may or may not include an introduction like the "Here are three…" row above. You can prevent this with more specific instructions passed to the function as additional messages such as "Return only the numbers as a markdown list" or even just "Format: markdown list" (or "Format: HTML list").

Without naming a specific list format, you can specify that you do not want the results as a CSV list on a single line by adding a message such as "Return one number per line".

3.9.2.4 `gpt:table()` — Retrieve a table of answers

This function sends one or more requests to create multiple bindings with multiple values — in other words, a table. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response.

Because `gpt:table` is a magic predicate, you call it as a triple pattern with one or more variables in which to store the response as the subject and the messages to pass as the object. The subject can list more than one variable to store the values of the table columns. If you have more than one subject variable, enclose the list in parentheses. Similarly, if you pass more than one message, enclose the list in parentheses. The following shows the potential forms:

```
?column1 gpt:table ?message
?column1 gpt:table (?message1 ?message2 ...)
(?column1 ?column2) gpt:table ?message
(?column1 ?column2 ?column3) gpt:table (?message1 ?message2 ...)
```

The following SPARQL query shows an example:

```
PREFIX gpt: <http://www.ontotext.com/gpt/>

SELECT * WHERE {
    (?name ?birthday ?instrument) gpt:table ("List the Beatles, their birthdays, and the instrument
↪that each played." )
}
```

The following shows the result:

	name	birthday	instrument
1	"John Lennon"	"October 9, 1940"	"Guitar, Vocals, Piano, Harmonica"
2	"Paul McCartney"	"June 18, 1942"	"Bass, Vocals, Guitar, Keyboards"
3	"George Harrison"	"February 25, 1943"	"Guitar, Vocals, Sitar, Keyboards, Bass"
4	"Ringo Starr"	"July 7, 1940"	"Drums, Percussion, Vocals"

In addition to having `gpt:table` create a new table for you, you can use the helper functions described at *List manipulation extension functions* with a `VALUES` clause to pass an incomplete table and ask the GPT model to fill in the missing values. Note how the three `undef` values in this next query's `VALUES` table get replaced by appropriate descriptions in the result:

```
PREFIX helper: <http://www.ontotext.com/helper/>
PREFIX gpt: <http://www.ontotext.com/gpt/>

SELECT * WHERE {
    {
        select (helper:tupleAggr(?row) as ?table) {
            values (?item ?description) {
                ("banana" "A banana is a curved, yellow fruit with a sweet taste and creamy texture.")
                ("strawberry" "A strawberry is a small, sweet, red fruit with a juicy texture and a
↪white center.")
```

(continues on next page)

(continued from previous page)

```
                ("pineapple" undef)
                ("egg" undef)
                ("mulberry" undef)
            }
            bind(helper:tuple(?item, ?description) as ?row)
        }
    }

    (?item ?description) gpt:table ("complete the missing columns" ?table)
}
```

	table	item	description
1	_:values-tuple1559832105	"Banana"	"A curved, yellow fruit with a sweet taste and creamy texture."
2	_:values-tuple1559832105	"Strawberry"	"A small, sweet, red fruit with a juicy texture and a white center."
3	_:values-tuple1559832105	"Pineapple"	"A large, tropical fruit with spiky, tough skin and sweet, juicy yellow flesh."
4	_:values-tuple1559832105	"Mulberry"	"A small, dark purple fruit with a sweet and tangy flavor."
5	_:values-tuple1559832105	"Egg"	"An oval or spherical object produced by female birds, reptiles, and fishes."

3.9.3 List manipulation extension functions

The GraphDB extensions to the SPARQL standard that are described in this section make it easier to assemble several pieces of data into a single structure to pass to *GPT functions*. (These extension functions are not related to the *Jena list function extensions* or to the collections predicates that can be used to create and manipulate RDF lists.)

3.9.3.1 `helper:tuple()` — Combine values into a list

This function combines all of its arguments into an internal list that you can reference using a blank node connected to the list members. You can access individual members of the list using the `helper:iterate()` function described further in *helper:iterate() — Iterate through an internal list*.

In the example below, the `helper:tuple()` function combines the two strings “foo” and “bar” and the IRI `ex:baz` into a list stored with the variable `?tuple`.

```
PREFIX helper: <http://www.ontotext.com/helper/>
PREFIX ex:     <http://www.example.com/>

SELECT ?tuple ?listMember WHERE {
  BIND(helper:tuple("foo", "bar", ex:baz) as ?tuple)
  ?listMember helper:iterate ?tuple .
}
```

In the result, we see that the `helper:iterate` magic predicate lists the `?tuple` members next to the identifier for the list’s blank node:

	tuple	listMember
1	_:values-tuple1560893376	"foo"
2	_:values-tuple1560893376	"bar"
3	_:values-tuple1560893376	http://www.example.com/baz

You can also call `helper:tuple` as a magic predicate. The following query does the same thing as the previous one, but using this function as a magic predicate in a triple pattern:

```
prefix helper: <http://www.ontotext.com/helper/>
PREFIX ex:     <http://www.example.com/>

SELECT ?tuple ?element WHERE {
    ?tuple helper:tuple ("foo" "bar" ex:baz) .
    ?element helper:iterate ?tuple
}
```

3.9.3.2 `helper:tupleAggr()` — Aggregate values into a list

The `helper:tupleAggr()` function is similar to the *helper:tuple* function but only takes one argument. If the argument is bound to multiple values, it will aggregate those values into a list that can be accessed by a blank node connected to the list members. You can access individual members of the list using the `helper:iterate()` function described further in *helper:iterate() — Iterate through an internal list*.

Note: Remember that `helper:tupleAggr()` is a SPARQL aggregate function and cannot be called inside of a `WHERE` clause like the functions that you might use with a `FILTER` clause.

In the example below, the `helper:tupleAggr()` function combines the values of all of the unique `?grapeName` values from the `wine dataset` into a list stored in the variable `?tuple`:

```
PREFIX wine: <http://www.example.com/demo/wine#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX helper: <http://www.ontotext.com/helper/>

SELECT ?tuple ?listMember WHERE {
  ?listMember helper:iterate ?tuple
  {
  SELECT (helper:tupleAggr(DISTINCT ?grapeName) as ?tuple) {
    ?grape a wine:Grape ;
           rdfs:label ?grapeName .
    }
  }
}
```

In the result, we see that the `helper:iterate` magic predicate lists the `?tuple` members next to the identifier for the list's blank node:

	tuple	listMember
1	_:values-tuple974022751	"Merlot"
2	_:values-tuple974022751	"Cabernet Sauvignon"
3	_:values-tuple974022751	"Cabernet Franc"
4	_:values-tuple974022751	"Pinot Noir"
5	_:values-tuple974022751	"Chardonnay"

3.9.3.3 `helper:rdf()` — Combine values into an RDF triple

The `helper:rdf()` function takes a subject, predicate, and object and combines them into an internal triple connected by a blank node that can be passed to the *helper:serializeRDF* function. You can provide the triple components directly or combine them using the *helper:tuple* function. If you do use the `helper:tuple` function, you may also add a reference to a named graph to store the triple as a quad as shown in the third form below:

```
helper:rdf(<< ?s ?p ?o >>)
helper:rdf(helper:tuple(?s, ?p, ?o))
helper:rdf(helper:tuple(?s, ?p, ?o, ?g))
```

One use case where this is useful is to query your own graph for the data that you are interested in and then pass that data to a GPT model with a natural language query about that data. The following query does this with these three steps:

1. Select the name and height values of the humans in a `Star Wars dataset`.
2. Use `helper:rdf` to bind these triples to the `?rdf` variable.
3. Pass the `?rdf` data with a natural language query about the height of certain characters to the GPT model using the *gpt:ask()* function:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX voc: <https://swapi.co/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX helper: <http://www.ontotext.com/helper/>
PREFIX gpt: <http://www.ontotext.com/gpt/>
SELECT ?answer WHERE {
    {
        SELECT (helper:rdf(helper:tuple(?s, ?p, ?o)) AS ?rdf) {
            ?s a voc:Human ;
                ?p ?o .
            FILTER(?p in (rdfs:label, voc:height))
        }
    }
    ?answer gpt:ask ("who is taller than 190cm or shorter than 170cm in this RDF data?" ?rdf)
}
```

	answer
1	"The characters who are taller than 190cm are Qui-Gon Jinn, Darth Vader, and Bail Prestor Organa. The characters who are shorter than 170cm are Mon Mothma, Padmé Amidala, Shmi Skywalker, and Leia Organa."

You can use the *helper:serializeRDF* extension function to learn more about what the `helper:rdf` function is storing as you refine your query.

3.9.3.4 `helper:serializeRDF()` — Convert internal RDF to readable triples

This function serializes RDF that has been stored internally with a function such as *helper:rdf* so that you can see the contents of the triples. The default format is Turtle, but you can request a different one by naming the MIME type as an optional second argument.

```
helper:serializeRDF(?rdf)
helper:serializeRDF(?rdf, "application/rdf+xml")
```

The following query is a modified version of the one shown in *helper:rdf() — Combine values into an RDF triple*. For this one, the `helper:serializeRDF()` function also stores a converted version of `?rdf` in the `?rdfSer` variable, and the outer `SELECT` statement requests the value of both with the answer.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX voc: <https://swapi.co/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX helper: <http://www.ontotext.com/helper/>
PREFIX gpt: <http://www.ontotext.com/gpt/>
SELECT ?rdf ?rdfSer ?answer WHERE {
    {
        SELECT (helper:rdf(helper:tuple(?s, ?p, ?o)) AS ?rdf) WHERE {
            ?s a voc:Human ;
                ?p ?o .
            FILTER(?p in (rdfs:label, voc:height))
        }
    }
    BIND(helper:serializeRDF(?rdf) as ?rdfSer)
    ?answer gpt:ask ("who is taller than 190cm or shorter than 170cm in this RDF data?" ?rdf)
}
```

In the result (of which only the beginning is shown below, because many more triples were serialized) we see that `?rdf` is the value of the blank node tying the triple components together and `?rdfSer` shows the Turtle-star serialization of the RDF.

	rdf	rdfSer	answer
1	_:rdf-value837286742	"<https://swapi.co/resource/human/1> <http://www.w3.org/2000/01/rdf-schema#label> "Luke Skywalker"; <https://swapi.co/vocabulary/height> "172.0"^^<http://www.w3.org/2001/XMLSchema#float> . <https://swapi.co/resource/human/10> <http://www.w3.org/2000/01/rdf-schema#label> "Obi-Wan Kenobi"; <https://swapi.co/vocabulary/height> "182.0"^^<http://www.w3.org/2001/XMLSchema#float> . <https://swapi.co/resource/human/11> <http://www.w3.org/2000/01/rdf-schema#label> "Anakin Skywalker"; <https://swapi.co/vocabulary/height> "188.0"^^<http://www.w3.org/2001/XMLSchema#float> .	"The characters who are taller than 190cm are Qui-Gon Jinn and Darth Vader. The characters who are shorter than 170cm are Mon Mothma, Padmé Amidala, Shmi Skywalker, Leia Organa, and Jocasta Nu."

3.9.3.5 `helper:iterate()` — Iterate through an internal list

This magic predicate iterates over the elements of an internal list created by the `helper:tuple`, `helper:tupleAggr`, or `helper:rdf` functions. See the sections *helper:tuple() — Combine values into a list* and *helper:tupleAggr() — Aggregate values into a list* for two examples.

3.9.4 GPT query explanations

GraphDB's *Query profiling with the Explain plan* feature lets you gather data about how a given query will execute so that you can explore ways to improve it. The GPT Explain feature gives you a similar way to learn more about what the query does and what the result represents.

The following query asks for the name of each Academy Award that the "Star Wars" movie won and how many people shared the award. If you run this query in the Workbench by holding down the `Alt` key when you click the Workbench's *Run* button, the result will include an additional `__gpt` column that provides information about both the query and the result, as shown below.

```
SELECT ?awardName (COUNT(?person) AS ?teamCount)
WHERE {
    ?award a voc:AwardRecognition ;
           voc:awardStatus "awarded" ;
```

(continues on next page)

(continued from previous page)

```
            voc:award ?awardType ;
            voc:person ?person ;
            voc:forWork "Star Wars"@en .
    ?awardType rdfs:label ?awardName .
    FILTER ( lang(?awardName) = "en" )
}
GROUP BY ?awardName
ORDER BY DESC(?teamCount)
```

	awardName	teamCount	__gpt
1			"The SPARQL query retrieves the count of team members for various awards received for the movie "Star Wars". The results are grouped by the name of the award. The result includes the award name and the corresponding count of team members. The results indicate that the award "Academy Award for Best Production Design" and "Academy Award for Best Film Editing" were awarded to a team of 6 people each. The "Academy Award for Best Visual Effects" was awarded to a team of 5 people. Similarly, the "Academy Award for Best Sound Mixing" had a team of 4 members. The "Academy Award for Best Original Score" and "Academy Award for Best Costume Design" both had 2 team members. Lastly, the "Special Achievement Academy Award" was awarded to a team of 1 member. The results are sorted in descending order based on the team count, showing the awards with the highest number of team members at the top."
2	"Academy Award for Best Production Design"@en	"6"^^xsd:integer	
3	"Academy Award for Best Film Editing"@en	"6"^^xsd:integer	
4	"Academy Award for Best Visual Effects"@en	"5"^^xsd:integer	
5	"Academy Award for Best Sound Mixing"@en	"4"^^xsd:integer	
6	"Academy Award for Best Original Score"@en	"2"^^xsd:integer	
7	"Academy Award for Best Costume Design"@en	"2"^^xsd:integer	
8	"Special Achievement Academy Award"@en	"1"^^xsd:integer	

For a `CONSTRUCT` or `DESCRIBE` query, doing this returns an extra triple with the output in the object: `onto:gpt onto:gpt "output"`.

Note: In addition to holding down the `Alt` key when you click the *Run* button, there are two other ways to add the `__gpt` column that describes the query and its result:

- By pressing `Alt+[Ctrl/Cmd]+Enter` with your cursor in the query edit field.
- By using `FROM <http://www.ontotext.com/gpt>` to add this special named graph to the query. This is useful when using the RDF4J Java client or the RDF4J REST API to send a query to GraphDB or to improve query reproducibility–for example, by using a Saved Query in GraphDB Workbench, sharing a query by email, or as a link.

3.9.4.1 Customizing the explanation output

Specialized strings added to your query as comments give you greater control over the output of a GPT query explanation.

- `:gpt: <additional instruction>` — Follow this additional instruction about how to provide the result. See example below.
- `:gpt-query-only: [optional instruction]` — Provide an explanation of the query but not the result. This can include an optional instruction to do something else with the query instead of explaining it.
- `:gpt-result-only: [optional instruction]` — Provide an explanation of the result but not the query. This can include an optional instruction to do something else with the result instead of explaining it.
- `:gpt-no-eval:` — Do not include the query result in the output.

The following query includes a `:gpt:` comment between the query's `SELECT` and `WHERE` clauses. Remember to execute it using the `Alt` key when you click the *Run* button or one of the alternatives to this described above:

```
PREFIX gpt: <http://www.ontotext.com/gpt/>

SELECT *
# :gpt: Respond in Shakespearean language
WHERE {
   ?primes gpt:ask "List three prime numbers that are greater than 100."
}
```

The result's `__gpt` explanation uses vocabulary and grammar similar to that of William Shakespeare:

	primes	__gpt
1		"Verily, the SPARQL query hath sought to uncover prime numbers greater than 100. The result doth proclaim that the numbers 101, 103, and 107 are such primes."
2	"101, 103, 107"	

Certain combinations of these instructions can work together. For example, the comments in the following DESCRIBE query (using the `Star Wars dataset`) tell GraphDB to ask the GPT model to write a song, and to not show the results of the query:

```
# :gpt-result-only: Write a short pop song about this data
# :gpt-no-eval:
DESCRIBE <https://swapi.co/resource/planet/1> <https://swapi.co/resource/human/1>
```

	subject	predicate	object
1	http://www.ontotext.com/gpt	http://www.ontotext.com/gpt	"(Verse 1) In a galaxy far away There's a planet called Tatooine A desert world with a climate arid Where humans and creatures convene (Chorus) Tatooine, Tatooine A planet of sand and dreams With binary sunsets that gleam Tatooine, Tatooine (Verse 2) Luke Skywalker, a hero's tale From Tatooine, he set sail A Jedi in training, his destiny awaits Fighting for freedom, against the empire's gates (Chorus) Tatooine, Tatooine A planet of sand and dreams With binary sunsets that gleam Tatooine, Tatooine (Bridge) On Tatooine, where the terrain's all desert Life thrives in the heat, a place of worth From droids to humans, they call it home A universe of stories, on Tatooine they're known

3.9.4.2 Setting the temperature of the explanation

A comment with `:gpt-temp: <number>` will set the temperature of the explanation. As with setting the *temperature* of a query result, this is a value between 0 and 2, with higher values requesting more randomness in the output. 1 is the default.

3.10 Talk to Your Graph

Note: This feature is currently still experimental.

Talk to Your Graph is a chatbot that lets you converse with your data and extract factual information using natural language. The chatbot is an example of Graph Retrieval-Augmented Generation (Graph RAG), as it retrieves relevant information from a GraphDB knowledge graph in the form of triples and uses that information to generate informed responses.

Talk to Your Graph provides the accuracy and depth of graph queries without requiring comprehensive understanding of SPARQL or other retrieval methods. It works with datasets of different sizes and can be tailored to different user-specific needs. The chatbot also stores information on the server side, allowing you to manage conversations between multiple interactions. The chatbot works with most large language models (LLMs) that are compatible with the Chat Completions API, and offers legacy support for OpenAI's Assistants API.

The chatbot builds on the concept of *agents* — individual assistants that can be configured to use different models, retrieval methods, and other settings. You can express additional configurations as written text to offer further context or instructions to the agent. (More information on creating and configuring Talk to Your Graph agents is available in the *Talk to your graph agent configuration parameters* section below.)

The chatbot uses several extraction methods — including SPARQL queries, *similarity search*, simple *full-text search*, and the *ChatGPT Retrieval Connector* — to request information from your knowledge graph, based on your query.

Once you've set up your agents, you can then use natural language to retrieve information from your knowledge graphs using the retrieval methods you have specified. Talk to Your Graph can also show you what specific methods and queries were used for each individual response, and it can perform regular actions found in most GPT interfaces such as regeneration of the response or copying it to the clipboard.

See also:

The two *Talk to Your Graph examples* showcase how to create and query a Talk to Your Graph agent using a simple dataset (*Star Wars example*) or a more complex, business-oriented dataset (*Acme employees and products example*).

Note: An older version of Talk to Your Graph existed prior to GraphDB 10.8 which used only the ChatGPT Retrieval Connector. For all practical purposes, that previous implementation is completely different from the current version of Talk to Your Graph.

3.10.1 Prerequisites and configuration

Before using Talk to Your Graph you must configure an LLM API type, an API key, and a default model:

- `graphdb.llm.api`: The address for the LLM API type that will determine the available LLM models.
- `graphdb.llm.api-key`: The authentication token for the LLM API.
- `graphdb.llm.model`: The model which Talk to Your Graph agents will use by default, if not changed during the creation process.

If an LLM API type is not provided with `graphdb.llm.api`, Talk to Your Graph will default to OpenAI's Assistants API. If the `graphdb.llm.api` property is set, `graphdb.llm.model` is required.

See also:

The full list of relevant configuration properties, as well as detailed explanation of which properties are required, is available in the *Talk to Your Graph properties* documentation.

3.10.1.1 Model Requirements

Talk to Your Graph uses large language models to process natural language questions and fetch relevant data from GraphDB using the provided extraction method. The response from GraphDB is then once again processed by the LLM, and returned as a natural language response to the user. Thus Talk to Your Graph should work with most LLM APIs which are compatible with the Chat Completions standard, and meet the following requirements:

- Must support function calling (tool calls)
- Must support multi-turn conversations
- Must be capable of generating code (specifically SPARQL queries)
- Should have a large context window to process the ontology and the memory

Note: The LLM needs the ontology of the knowledge graph to succeed in generating a truthful response. Different models may be suitable for different ontologies. We strongly recommend experimenting with different models (if possible) to see which one yields the best results for your specific ontology.

Warning: Small local models often do not perform well in SPARQL query generation. However, a bigger model does not necessarily translate to better results, and must be tested first.

3.10.2 Talk to Your Graph query methods

When *creating a Talk to Your Graph agent*, you can choose any combination of the four available query methods to retrieve information from the dataset. Each method is specialized to answer different types of questions and work with different types of datasets. Each agent must have one or more of the query methods enabled.

Query method	Requirements	Description
SPARQL query	One of the following: • The named graph that contains the data's class and property definitions stored as an ontology or schema. • A SPARQL query that fetches ontology or schema information from the repository.	Answers questions by performing a SPARQL query. Performs well on close-ended questions, including aggregations. Suitable for datasets with well-defined *schemas* or *ontologies*.
Full-text search	*Full-text search* must be enabled at repository configuration level.	Answers questions by using full-text search in literals and IRIs. Performs well on open-ended questions and in datasets rich in text data, but underperforms when providing aggregations.
Semantic similarity search	The tool can now use either: • A *similarity index* • A connector for similarity search (*Elasticsearch* or *OpenSearch*) that has declared at least one field with datatype `"vector"` or `"native:semantic_text"` (if using Elasticsearch)	Answers questions by using Similarity search. Performs well on open-ended questions, but underperforms when providing aggregations.
ChatGPT Retrieval connector	*ChatGPT retrieval connector* must be created.	Answers questions by using vector search in text chunks created from RDF. Performs well on open-ended questions, but underperforms when providing aggregations.

Note: When using full-text search, it is a good idea to limit the number of results to prevent results from filling the context window. This is especially advisable when using Talk to Your Graph with models with smaller context windows.

You can also set the full-text search query itself using the configuration property `graphdb.ttyg.fts.query.template`. Check the *Talk to Your Graph properties* documentation for more information.

Tip: An agent that combines *SPARQL query* with either *Full-text search* or *Semantic similarity search* is suitable for most use cases, as it will be able to adequately answer both close-ended and open-ended questions and switch between methods based on the type of question the user asks.

A hidden query method called the Now function is always enabled and provides the current date and time as its output. The result is in the time zone of the user browser that initiated the chat request that led to the model calling this tool. It has no settings. This tool helps the model answer questions that involve relative time such as "how many new employees were hired this year" or "is it OK to call John at this time?" (presuming the model can find out John's time zone and what are appropriate hours to call him), or even things as simple as "what time is it?"

3.10.2.1 IRI discovery query methods

You can also enable additional query methods — *Autocomplete for IRI discovery* and *Full-text search in labels for IRI discovery* — as a supplement to the main query methods listed above. These methods help the model discover the IRIs for a particular incomplete query when it cannot answer general questions about the dataset on its own.

We recommend you try *Autocomplete for IRI discovery* first. It requires autocomplete for the repository to be enabled and comes with default values for max number of results (IRIs) per call which can be individually configured.

3.10.3 Create and configure a Talk to Your Graph agent

Note:

- Only users with a repository manager role can create agents. However, any user that has access to a repository also has access to the agents created for that repository.
- Talk to Your Graph does not support *Ontop* and *FedX* repositories.

To create your first agent, select *Lab* ▸ *Talk to Your Graph* on the GraphDB main menu, click on *Create your first agent*, and provide the necessary configuration parameters. (If you already have one or more agents, click on the *Create a new agent* ⊕ icon on the top right of the Talk to Your Graph interface.)

3.10.3.1 Talk to Your Graph agent configuration parameters

The Talk to Your Graph agents take several configuration parameters:

- *Agent name*: The name of your agent.
- *Repository ID*: The repository for which you are creating the agent.
- *Query methods*: Allows you to enable or disable each of the four *query methods* individually. When enabled, each method also takes additional configuration parameters:
  - *SPARQL*: When querying with SPARQL, you will need an ontology. You can set this to fetch the ontology from a specific named graph stored with the repository named in *Repository ID*. Instead of fetching the ontology from a named graph, you can also use a `CONSTRUCT` query to identify the triples in that repository that serve as an ontology. When using a SPARQL query method, you can use the *Auto-add missing namespaces* checkbox to tell Talk to Your Graph to automatically insert namespace prefix declarations when a query uses prefixes without declaring them if Talk to Your Graph can find those declarations in the repository.

    See also:

    Loading data using the Workbench has a section on the import settings; this outlines how to load data such as an ontology into a named graph within a repository.

    If you do select *Fetch ontology from a named graph*, instead of naming a specific named graph you can use one of several special IRIs that are hints to insert a message in the instructions that tell the model what ontology to use. The following shows the supported values and what ontology schemas they tell the model to use in SPARQL queries:

    * `onto:use:schema.org` the schema.org ontology
    * `onto:use:foaf` the FOAF ontology
    * `onto:use:wikidata` the Wikidata ontology (note that GraphDB cannot handle Wikidata SPARQL extensions)
    * `onto:custom/<some text here>` <some text here>

- *Full-text search*: You can set the maximum number of results per call. If left unchanged, the maximum number of triples will be automatically determined.

  Note: To use this method, you must enable the full-text search index on the corresponding repository.

- *Similarity search*: Requires the name of the Similarity or connector index. Takes the maximum number of results per call. If the latter is left unchanged, the maximum number of triples will be automatically determined.

  This method takes several additional configuration parameters of its own:

Parameter	Type	Description	Default value
`similarityIndex`	`string`	Name of the similarity index (for Similarity plugin index) or connector instance for similarity search (for Elasticsearch/OpenSearch connectors)	
`resultThreshold`	`double`	Minimum similarity score for results to be included	`0.4`
`limit`	`int`	Maximum number of triples to return per call (`0` is unlimited)	`0`
`connectorType`	`string`	Determines which backend is used: `"similarity"`, `"elasticsearch"`, or `"opensearch"`	
`connectorFields`	`array of string`	(For connectors only) Specifies which vector field (one) to use in the search query	

  The agent uses only *text-based similarity search*, and does not support the use of *predicate-based similarity indexing* or other methods.

  Note: To use this method, you must create a connector with a vector field (*Elasticsearch* or *OpenSearch*) or a similarity search index.

- *ChatGPT Retrieval Connector*: Requires the ChatGPT retrieval connector ID. You can edit the query template and set a maximum number of results (chunks) per call. If the latter is left unchanged, the maximum number will be automatically determined.

  Note: To use this method, you must create a ChatGPT retrieval connector.

- *IRI discovery query methods*: Described under *Talk to Your Graph Query Methods*.
- *Model*: Sets the LLM model of the agent. If this is left unchanged, the agent will use the default model, as described in *Prerequisites and configuration*.

  Tip: If you are unsure what models are available for your deployment, your administrator should be able to provide you with a list.

- *Temperature*: Sets the sampling temperature of the agent. Lower values result in more consistent, repetitive responses generated by the agent and higher values result in a wider variety of responses. This takes a value

between 0 and 2, but values higher than 1 may result in unreliable and unpredictable answers and are not recommended.

- *Context window size*: Sets the maximum size of the context sent to the model with each request, measured in tokens. When the context window exceeds the maximum size, the mechanism will start removing old messages to fit the maximum size.

  The default size is 1,000,000 and is controlled by the `graphdb.ttyg.model.context.window.size` property.

  > **Warning:** The maximum context window size **must not** be bigger than what the model supports. If the maximum size exceeds that, it will eventually result in errors when using the agent. Consult with the documentation of the model you are using to check what the model supports.

  Note: The context window size parameter is not configurable when using the ChatGPT Assistants API.

- *Top P*: Sets the nucleus sampling of the agent. Upon retrieving a set of candidate values, this controls how many of those values to consider using. Takes a value between 0 and 1.
- *Additional instructions*: Provides additional instructions and context, such as general information about the dataset, to the agent as written text.

Advanced settings:

- *Base instructions*: The instructions that provide the agent with a personality, as well as general context on answering questions about data stored in GraphDB.

  > **Warning:** Changing the base instructions is is not recommended and may hurt performance.

After configuring your new agent, click the *Create Agent* button in the lower right. Next, select your agent from the drop-down list at the top and you can start entering natural language queries for that agent in the *Ask* field at the bottom of your screen. For example, if you graph has information about employees, you could enter "Which employee was hired the most recently?"

See also:

- *The Talk to Your Graph configuration properties in GraphDB*

3.10.3.2 Manage Talk to Your Graph agents

Once you have created one or more agents, you can expand or hide the Manage agents panel by clicking on the *Manage agents* icon on the top right of the Talk to Your Graph interface. Once expanded, the panel will display a list of all available agents. You can then filter agents by repository by choosing a repository from the drop-down menu on the top of the agents list.

Manage agent settings

1. Click the *Manage agents* icon and scroll to the name of the agent whose settings you want to edit.
2. Click on the options ··· icon next to an agent's name.
3. Click on *Settings*.
4. Edit the settings of the agent.

Note: You can also change the repository from which the agent retrieves information. Doing so will not affect previous messages in existing conversations with the agent, but if you continue such an existing conversation, the agent will answer any new queries using the newly selected repository.

Clone agent

1. Click the *Manage agents* icon and scroll to the name of the agent that you want to clone.
2. Click on the options ··· icon next to the agent's name.
3. Click on *Clone*.
4. Edit the settings of the new agent.

Delete agent

1. Click the *Manage agents* icon and scroll to the name of the agent that you want to delete.
2. Click on the options ··· icon next to the agent's name.
3. Click on *Delete*.

Note: This will not remove your chat history with that agent. However, you will be unable to continue the conversation unless you select a new agent.

3.10.4 Using the Talk to Your Graph chat interface

The chat is the main interface for asking questions and receiving responses from Talk to Your Graph. There are two ways to start a conversation:

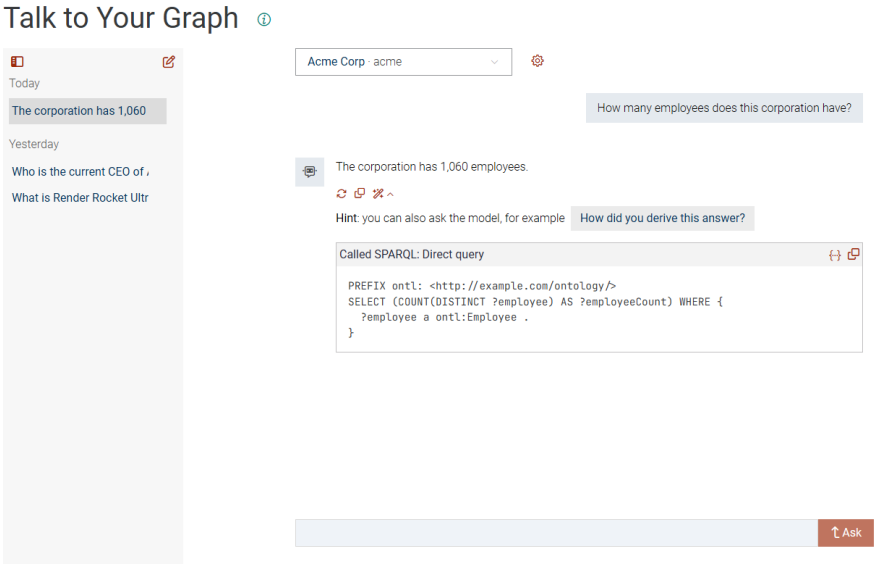
- If there aren't any conversations available yet, select an agent if there isn't one selected already, enter a natural language query in the *Ask* field (for example, "What kind of data do you have?") and click its button. The chat will be automatically created and will appear in the chat list on the left.
- Click the *Start a new chat* icon at the top of the chat list. This creates a new entry on the list and puts your cursor in the *Ask* field. The most recently used agent from a previous chat session may be automatically selected; if not, select one from the drop-down list at the top of the Talk to Your Graph screen.

If you leave and return to the Talk to Your Graph page in your browser, the most recently used agent and chat will be already selected.

Warning: You can also cancel a running question — for example, when an agent takes too long to respond. However, be aware that this does not guarantee no tokens are consumed. Tokens may still be spent during processing, and the exact amount used cannot be predicted in advance. Use caution when canceling queries, particularly those involving large or complex datasets.

3.10.4.1 Explaining agent responses

You can ask the agent to explain how it obtained the information for any of its responses. Clicking on the *Explain response* icon under any response will prompt the model to respond with the query methods used. It may display a suggested prompt (like "How did you derive this answer?" below) that you can then click on to execute.



When it displays a SPARQL query, the *Open in SPARQL editor* icon in the upper right of the SPARQL window copies the query to Workbench's SPARQL screen where you can explore variations on the query and execute them. The *Copy SPARQL query* icon to the right of that copies the query into your clipboard.

3.10.4.2 Managing chats

Talk to Your Graph remembers each user's chats and can display a list of them grouped by date on the left of the Talk to Your Graph screen, where you can manage them. Each chat can use multiple repositories and multiple agents. When you select a chat from the list, Talk to Your Graph loads it so that you can resume that chat using the agent that was most recently used with it.

In the following the chat list has three chats under "Today". The mouse pointer is hovering to the right of the third one, displaying a menu for actions you can take on that chat.

The *Hide/Show chat list* and *Start a new chat* icons above the list are self-explanatory. Entering a question in the *Ask* field at the bottom will also create a new chat if none currently exists.

A chat's name on the list will be the beginning of the first response from the agent in that chat. As shown above, the *Rename* choice of the pop-up menu with each chat name will let you rename it; double-clicking a name on the

list will also let you do this.

The *Delete* choice on that menu removes that chat, and the *Export* choice downloads the chat in JSON format.

3.11 Graph path search

3.11.1 Overview

The GraphDB Graph path search feature allows you to not only find complex relationships between resources but also explore them and use them as filters to identify graph patterns. This is a key factor in a variety of use cases and fields such as data fabric analysis of supply chains, clinical trials in drug research and social media management. Discovering connections between resources must come hand in hand with the ability to explain them to key stakeholders.

It includes algorithms for Shortest path and All paths search, which enable you to explore the connecting edges (RDF statements) between resources for the shortest property paths and subsequently for all connecting paths. Other supported algorithms include finding the shortest distance between resources and discovering cyclical dependencies in a graph.

Graph path search also supports wildcard property search and more targeted graph pattern search. A graph pattern is an edge abstraction that can be used to define more complex relationships between resources in a graph. It targets specific types of relationships in order to filter and limit the number of paths returned. For example, it can define indirect relationships such as N-ary relations that rely on another resource and that cannot be expressed using a standard subject-predicate-object directional relationship.

The graph path search extension is compatible with the GraphDB service plugin syntax, which allows for easy integration into queries.

Tip: Graph path search is similar to the SPARQL 1.1 property paths feature. Both enable graph traversal, letting you discover relationships between resources through arbitrary length patterns. However, property paths uncover the start and end nodes of a path, but not the intermediate ones, meaning that traceability remains a challenge.

For the examples included further down in this page, we have used a dataset containing Marvel Studios-related data combined with some information from DBpedia. To try them out yourself, download the `sample Marvel Studios data` it and load it into a new GraphDB repository via *Import* ▸ *User data* ▸ *Upload RDF files*.

3.11.2 Usage

Four graph path search algorithms are supported: *Shortest path*, *All paths*, *Shortest distance*, and *Cyclic path*.

For Shortest path and All paths, the following is valid:

- GraphDB returns search results for all of the paths with the the shortest length. If, when searching for the shortest path between two nodes, there are several different paths that meet this requirement, all of them will be returned as results.
- Bindings for at least the source or destination (preferably both) must be provided.
- The `startNode` and `endNode` properties are unbound prior to path evaluation and are bound by the path search for each edge returned by the query. If a graph pattern is used, they show the relation between the two nodes, and are bound by the path search dynamically and recursively.
- Edges can be returned as *RDF-star statements*.
- Each binding can also be returned separately.
- When using a wildcard predicate pattern, the edge label (predicate) can be accessed as well.

All of the graph path search algorithms support the use of a literal as a destination. Both source and destination can be literals (for example, N-ary relations).

`path:findPath` is a required property that defines the type of search function.

A graph path search is defined by three types of properties described in detail below.

Path Search Algorithms

Property name	Description
`path:shortestPath`	**Required** property that computes the shortest path between two input nodes or between one bound and one unbound node. If, when searching for the shortest path between two nodes, there are several different paths that meet this requirement, all of them will be returned as results.
`path:allPaths`	**Required** property that finds all paths between two nodes or between all nodes and the starting node.
`path:distance`	**Required** property that finds the distance of the shortest path between two resources, which is the number of edges that connect the resources.
`path:cycle`	**Required** property that finds cyclic dependencies for a given resource, meaning that a resource points back to itself.

Modifier Bindings

Property name	Description	Supported algorithms
`path:poolSize`	**Optional** modifier that enables parallel path search query evaluation. This parameter lets you specify the size of the thread pool used to evaluate the input path search query in parallel. It is limited by the total number of cores available per license — the more licensed cores, the larger the pool size and the faster the query execution. See also *Parallel search mode*.	• Shortest path • All paths • Shortest distance • Cyclic path

Variable Bindings

Property name	Description	Supported algorithms
`path:sourceNode`	**Required** variable binding that specifies the source node from which the path search commences. If a destination is selected, this variable can be optional.	• Shortest path • All paths • Shortest distance • Cyclic path
`path:destinationNode`	**Required** variable binding that specifies the destination node where the path traversal completes. If a source is selected, this variable can be optional.	• Shortest path • All paths • Shortest distance
`path:distanceBinding`	**Required** variable binding that returns the value of `path:distance`, without which the feature cannot be used. Cannot be added as property for any other type of path search.	• Shortest distance

Continued on next page

Table 4 – continued from previous page

Property name	Description	Supported algorithms
`path:resultBinding`	**Optional** variable binding used to view the path edges as RDF-star statements. If a wildcard predicate pattern is used, the actual properties connecting the resources inside the path would be fetched as well. If a graph pattern is used, they would not be accessible, and the magic predicate `path:connectedTo` would be returned.	• Shortest path • All paths • Cyclic path
`path:startNode`	**Optional** variable binding that specifies the starting resource in the recursive graph pattern. This variable should only be used when defining a graph pattern rather than using a wildcard predicate pattern. It returns the interim source node for each edge inside the path. This is equally useful with and without a graph pattern in cases where we do not care about the entire edge.	• Shortest path • All paths • Shortest distance • Cyclic path
`path:endNode`	**Optional** variable binding that specifies the ending resource in the recursive graph pattern. This variable should only be used when defining a graph pattern rather than using a wildcard predicate pattern. It returns the interim destination node for each edge inside the path. This is equally useful with and without a graph pattern in cases where we do not care about the entire edge.	• Shortest path • All paths • Shortest distance • Cyclic path
`path:propertyBinding`	**Optional** variable binding used to view the properties connecting the resources inside a path at each step. This variable can only be used with a wildcard predicate search.	• Shortest path • All paths • Cyclic path
`path:resultBindingIndex`	**Optional** variable binding that returns the index of each edge inside a path in incremental order. It follows the Java array indexing notation that starts from 0.	• Shortest path • All paths • Cyclic path
`path:pathIndex`	**Optional** variable binding that returns the index of each path returned by the graph path search in incremental order. For each path, all edges that constitute it will have the same path index. Path indexing follows the Java array indexing notation that starts from 0.	• Shortest path • All paths • Cyclic path
`path:exportBinding`	**Optional** variable binding that returns bindings from the graph pattern query service. Can also be specified in optional blocks, unions and so forth. Must be defined inside the main service of the path search query, and the names defined in the parameters of the search query must be present in the nested graph pattern service. Cannot be used with a wildcard predicate.	• Shortest path • All paths • Cyclic path
`path:bidirectional`	**Optional** variable binding that traverses adjacent nodes both in the S-P-O order and the O-P-S order where the subject and object are the recursively evaluated start and end nodes. Can also be specified together with export bindings.	• Shortest path • All paths • Shortest distance • Cyclic path

Filtering Parameters

Property name	Description	Supported algorithms
`path:minPathLength`	• **Optional** variable binding that specifies the minimal path length returned by an all paths search. This property is inclusive (meaning that a minimum length of 3 edges or edge abstractions for graph patterns would fetch all paths with length 3 and higher) and requires a value of type `xsd:int`. • The **default value** is -1, meaning that there is no minimal requirement.	• All paths • Cyclic path
`path:maxPathLength`	• **Optional** variable binding that specifies the maximum path length returned by an all paths or shortest path search. This property is inclusive (meaning that a maximum length of 3 edges or edge abstractions for graph patterns would fetch all paths with length 3 and less) and requires a value of type `xsd:int`. • The **default value** is 8, but if set to -1, the graph path search would fetch all the paths with no limit in terms of length.	• Shortest path • All paths • Shortest distance • Cyclic path

Required properties include a binding for source and/or destination, as well as the type of the search.

Optional properties include a minimum or maximum path length, edge bindings, or path indexing. Setting a maximum path length can be useful, for instance, when you are querying a large repository of over several hundred million statements and want to limit the results so as to not strain the database.

3.11.2.1 Search algorithms

GraphDB has several classic graph traversal algorithms built in so that you can call them from SPARQL queries.

Shortest path

This algorithm finds the shortest path between two nodes identified in its input or between one bound and one unbound node. It recursively evaluates the graph pattern in the query and replaces the start variable with the binding of the end variable in the previous execution. If we have specified a start node in the query, its value is used for the first evaluation of the graph pattern. If we have specified an end node, the query execution will stop when that end node is reached.

You can use the shortest path algorithm with a wildcard predicate as well as a graph pattern that is used as an edge abstraction. With it, we can impose filtering through techniques such as property negation or selection, define indirect relationships, and specify the use of named graphs.

Note: Inside the graph pattern, we cannot define other sub-queries or use federated queries for performance reasons. The variables bound as objects to the `path:startNode` and `path:endNode` properties must be present at least once inside the graph pattern.

See *Shortest path search examples*.

All paths

This algorithm finds all paths between two nodes or between all nodes and the starting/destination node. You can use it with a wildcard predicate as well as with more complex graph patterns and relationships. With it, we can also impose filtering with a minimum or maximum number of edges and can include or exclude inferred edges.

Tip: See *All paths search examples*.

Shortest distance

The algorithm finds the distance of the shortest path between two resources, which is the number of edges that connect the resources. This is done through the `path:distanceBinding` property. This returns the distance value and not the nodes themselves.

Tip: See a *Shortest distance search example*.

Cyclic path

With the cyclic path search we can explore self-referring relationships between resources. Similarly to the All paths search, this one can also be limited with minimum or maximum values.

Tip: See a *Cyclic path search example*.

3.11.2.2 Search modifiers

Parallel search mode

This mode enables parallel path search query evaluation and lets you specify the size of the thread pool used to evaluate in parallel the input path search query. It is limited by the total number of cores available per license — the more licensed cores, the larger the pool size and the faster the queries. It is very effective when used with complex graph patterns.

To perform parallel path search, use the `path:poolSize` global modifier property. The number of parallel threads used by all parallel path searches simultaneously cannot exceed the number of licensed cores.

Tip: See a *Parallel path search example*.

Exportable graph pattern bindings

Export bindings let you project any number of bindings from the graph pattern query service. This combines the power of SPARQL graph pattern-matching property paths with GraphDB's path search algorithm, letting the user restrict the start and the end nodes of the path search to those pairs that match a particular graph pattern defined as a SPARQL property path. You can "export" bindings from such graph patterns and this way get additional details about the found paths.

The export bindings as parameters must be defined inside the main service of the path search query with the magic predicate `<http://www.ontotext.com/path#exportBinding>` (or simply `path:exportBinding`). Keep in mind

that the binding names defined in the parameters of the search query must be present in the nested graph pattern service.

Tip: See an *Export bindings example*.

Bidirectional search

This can be used to traverse paths as if the graph is undirected (that is, that the edges between the nodes have no direction). Technically, bidirectional search traverses adjacent nodes both in S-P-O and O-P-S order, where the subject and object are the recursively evaluated start and end nodes. You can use it with all functions and can be combined with wildcard and graph pattern search as well as with exportable graph pattern bindings.

In order to do bidirectional search, you can use the magic predicate `<http://www.ontotext.com/path#bidirectional>` (or simply `path:bidirectional`) followed by the value `true` of type `xsd:boolean`.

Tip: See a *Bidirectional graph path search example*.

3.11.3 Usage examples

3.11.3.1 Shortest path

Let's try out the shortest path search with queries that we will run against the Marvel Studios dataset that we loaded into GraphDB earlier.

Shortest path search with a wildcard predicate

Suppose we want to find the shortest path between the movie **"The Black Panther (1977)"** as a source node and Marvel Comics' creative leader **Stan Lee** as the destination node.

In the Workbench SPARQL editor, run the following query:

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?pathIndex ?edgeIndex ?edge
WHERE {
    VALUES (?src ?dst) {
        ( dbr:The_Black_Panther_\(1977_film\) dbr:Stan_Lee )
    }
    SERVICE path:search {
        [] path:findPath path:shortestPath ;
           path:sourceNode ?src ;
           path:destinationNode ?dst ;
           path:pathIndex ?pathIndex ;
           path:resultBindingIndex ?edgeIndex ;
           path:resultBinding ?edge ;
           .
    }
}
```

Here, the path traversal is done with a wildcard predicate. This is because we want to explore the predicates connecting the resources inside the path, and we do not know the relationships within the data.

The `path:resultBinding` property returns path edges as RDF-star statements. Each edge is indexed with the `path:resultBindingIndex` property and each of the shortest paths is indexed with the `path:pathIndex` property.

The results show that there are ten shortest paths between Stan Lee and the 1977 “Black Panther” movie (paths 0-9), each consisting of four edges. The first one, for example, reveals the following relationship:

“The Black Panther (1977)” is a different movie from “Black Panther”. The studio that made “Black Panther” is Marvel Studios, founded by Marvel Entertainment, where Stan Lee is a key person.

	pathIndex	edgeIndex	edge
1	"0"^^xsd:int	"0"^^xsd:int	<< http://dbpedia.org/resource/The_Black_Panther_(1977_film) owl:differentFrom http://dbpedia.org/resource/Black_Panther_(film) >>
2	"0"^^xsd:int	"1"^^xsd:int	<< http://dbpedia.org/resource/Black_Panther_(film) http://dbpedia.org/property/studio http://dbpedia.org/resource/Marvel_Studios >>
3	"0"^^xsd:int	"2"^^xsd:int	<< http://dbpedia.org/resource/Marvel_Studios http://dbpedia.org/property/founder http://dbpedia.org/resource/Marvel_Entertainment >>
4	"0"^^xsd:int	"3"^^xsd:int	<< http://dbpedia.org/resource/Marvel_Entertainment http://dbpedia.org/ontology/keyPerson http://dbpedia.org/resource/Stan_Lee >>

The `pathIndex` column in the results shows which path the edge belongs to. Because only the first path of the returned result is shown above, all the edges have a `pathIndex` value of 0. The `edgeIndex` values show each edge’s position within that path.

We can also trace the path in Workbench’s *Visual graph* tool:

1. Go to *Setup* ▸ *Autocomplete* to enable it.
2. From *Explore* ▸ *Visual graph* ▸ *Easy graph*, search for the resource **The Black Panther (1977)** (the resource view will autocomplete the IRI).
3. Trace the identified path.

Note: Due to the large number of connections in the dataset and for better readability, in this and the following examples, the relationships in the Visual graph are filtered to display only the resources connected by the preferred predicates `differentFrom`, `studio`, `founder`, and `keyPerson`.

Shortest path search with graph pattern

This next query will also search for the shortest path between source node **"The Black Panther (1977)"** and destination node **Stan Lee**, but this time excluding any properties of the type `http://dbpedia.org/property/keyPerson`. The path traversal will be executed using a graph pattern specifying the exclusion of this property type through property negation with the SPARQL 1.1 property paths syntax.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT ?start ?end ?index ?path
WHERE {
    VALUES (?src ?dst) {
        ( dbr:The_Black_Panther_\(1977_film\) dbr:Stan_Lee )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:shortestPath ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:pathIndex ?path ;
                   path:startNode ?start;
                   path:endNode ?end;
                   path:resultBindingIndex ?index .
        SERVICE <urn:path> {
            ?start !dbp:keyPerson ?end
        }
    }
}
```

The paths are "served" by the nested `SERVICE <urn:path>` sub-clause where the service IRI coincides with the subject node invoking `path:findPath`. The paths connect the nodes specified by the `path:startNode` and `path:endNode` bindings.

As we are using a graph pattern to specify the relation, we cannot view the predicates connecting the resources. In other words, `path:resultBinding` is not applicable, but we can still view the nodes.

As in the previous example, we can index the edge bindings with the `path:resultBindingIndex` property and index each of the shortest paths with the `path:pathIndex` property.

After excluding the DBpedia `keyPerson` property from the search, two shortest paths between these resources are returned as results:

- “The Black Panther (1977)” – “Black Panther” – Marvel Studios – Marvel Entertainment – Stan Lee
- “The Black Panther (1977)” – “Black Panther” – Marvel Studios – Marvel Productions – Stan Lee

	start	end	index	path
1	http://dbpedia.org/resource/The_Black_Panther_(19	http://dbpedia.org/resource/Black_Panther_(film)	"0"^^xsd:int	"0"^^xsd:int
2	http://dbpedia.org/resource/Black_Panther_(film)	http://dbpedia.org/resource/Marvel_Studios	"1"^^xsd:int	"0"^^xsd:int
3	http://dbpedia.org/resource/Marvel_Studios	http://dbpedia.org/resource/Marvel_Entertainment	"2"^^xsd:int	"0"^^xsd:int
4	http://dbpedia.org/resource/Marvel_Entertainment	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"0"^^xsd:int
5	http://dbpedia.org/resource/The_Black_Panther_(19	http://dbpedia.org/resource/Black_Panther_(film)	"0"^^xsd:int	"1"^^xsd:int
6	http://dbpedia.org/resource/Black_Panther_(film)	http://dbpedia.org/resource/Marvel_Studios	"1"^^xsd:int	"1"^^xsd:int
7	http://dbpedia.org/resource/Marvel_Studios	http://dbpedia.org/resource/Marvel_Productions	"2"^^xsd:int	"1"^^xsd:int
8	http://dbpedia.org/resource/Marvel_Productions	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"1"^^xsd:int

In the Visual graph, it will look like this:

3.11.3.2 All paths

All paths search with unbound source

The next query will find **all resources and their respective paths** that can reach resource **Stan Lee** with a minimum of five edges using a wildcard predicate pattern.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?edge ?index ?path
WHERE {
    VALUES (?dst) {
        ( dbr:Stan_Lee )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
```

(continues on next page)

(continued from previous page)

```
                    path:destinationNode ?dst ;
                    path:pathIndex ?path ;
                    path:minPathLength 5 ;
                    path:resultBinding ?edge ;
                    path:resultBindingIndex ?index .
    }
}
```

As with Shortest path, path edges are returned as RDF-star statements through the `path:resultBinding` property. Each edge is indexed with the `path:resultBindingIndex` property.

The first returned path will be:

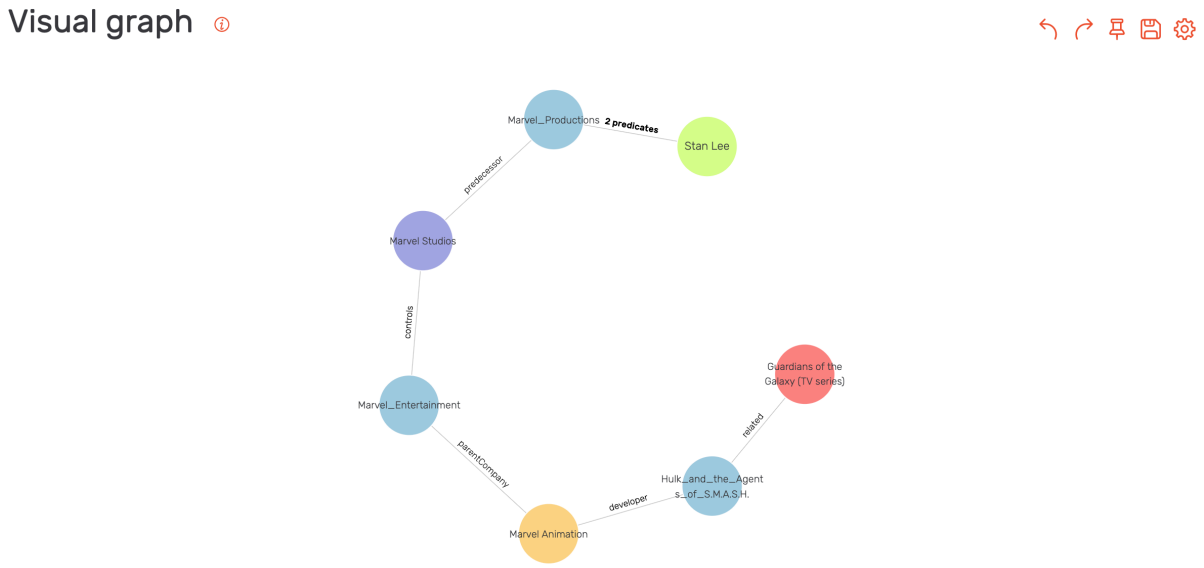
	edge	index	path
1	<< http://dbpedia.org/resource/Guardians_of_the_Galaxy_(TV_series) http://dbpedia.org/ontology/related http://dbpedia.org/resource/Hulk_and_the_Agents_of_S.M.A.S.H. >>	"0"^^xsd:int	"0"^^xsd:int
2	<< http://dbpedia.org/resource/Hulk_and_the_Agents_of_S.M.A.S.H. http://dbpedia.org/ontology/developer http://dbpedia.org/resource/Marvel_Animation >>	"1"^^xsd:int	"0"^^xsd:int
3	<< http://dbpedia.org/resource/Marvel_Animation http://dbpedia.org/ontology/parentCompany http://dbpedia.org/resource/Marvel_Entertainment >>	"2"^^xsd:int	"0"^^xsd:int
4	<< http://dbpedia.org/resource/Marvel_Entertainment http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/controls http://dbpedia.org/resource/Marvel_Studios >>	"3"^^xsd:int	"0"^^xsd:int
5	<< http://dbpedia.org/resource/Marvel_Studios http://dbpedia.org/property/predecessor http://dbpedia.org/resource/Marvel_Productions >>	"4"^^xsd:int	"0"^^xsd:int
6	<< http://dbpedia.org/resource/Marvel_Productions http://dbpedia.org/ontology/keyPerson http://dbpedia.org/resource/Stan_Lee >>	"5"^^xsd:int	"0"^^xsd:int

Visualizing path search results is possible through the CONSTRUCT query where you can propagate bindings from each edge through the `path:startNode`, `path:endNode`, `path:exportBinding` (for more complex traversals), and `path:propertyBinding` (when not specifying graph patterns) to the CONSTRUCT query projection.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

CONSTRUCT {
    ?start ?edgeLabel ?end
} WHERE {
    VALUES (?dst) {
        ( dbr:Stan_Lee )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:allPaths ;
                    path:sourceNode ?src ;
                    path:destinationNode ?dst ;
                    path:minPathLength 5 ;
                    path:startNode ?start ;
                    path:propertyBinding ?edgeLabel ;
                    path:endNode ?end ;
    }
}
```

With the *Visual* button now visible at the bottom right of the SPARQL editor, you can see the results in the visual

graph:



Warning: The graph visualization tool is not fully compatible with the graph path search feature and in most cases would not display every path returned by the path search query.

All paths search with unbound destination

The following query retrieves **all resources and their respective paths** that can be reached by the resource **"Guardians of the Galaxy (TV series)"** with a minimum of four and a maximum of five edges using a wildcard predicate pattern.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?start ?property ?end ?index ?path
WHERE {
    VALUES (?src) {
        (   dbr:Guardians_of_the_Galaxy_\(TV_series\))
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:minPathLength 4 ;
                   path:maxPathLength 5 ;
                   path:startNode ?start;
                   path:propertyBinding ?property ;
                   path:endNode ?end;
                   path:resultBindingIndex ?index ;
                   path:pathIndex ?path .
    }
}
```

All edge nodes as well as predicates connecting them are viewed through the `path:startNode`, `path:propertyBinding`, and `path:endNode` properties.

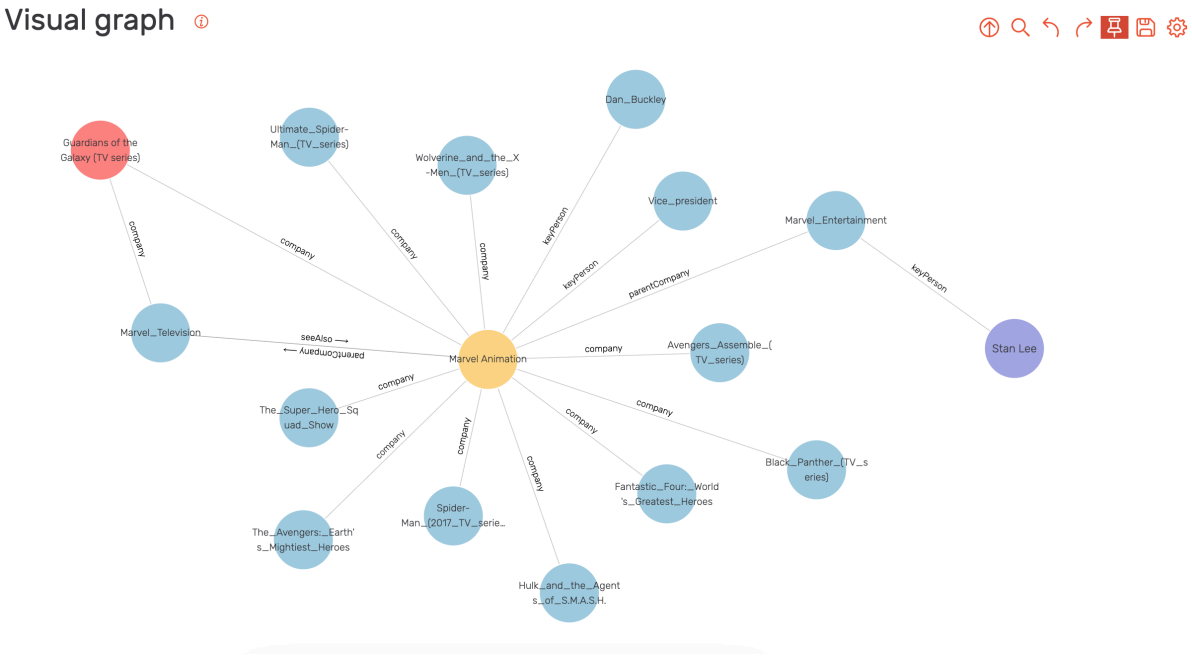
Tip: There are multiple ways to return results — for example, path edges returned as RDF-star statements through

the `path:resultBinding` property.

These will be the first four paths returned:

	start	property	end	index	path
1	http://dbpedia.org/resource/Guardians_of_the_Galaxy_(TV_series)	http://dbpedia.org/property/company	http://dbpedia.org/resource/Marvel_Television	"0"^^xsd:int	"0"^^xsd:int
2	http://dbpedia.org/resource/Marvel_Television	rdfs:seeAlso	http://dbpedia.org/resource/Marvel_Animation	"1"^^xsd:int	"0"^^xsd:int
3	http://dbpedia.org/resource/Marvel_Animation	http://dbpedia.org/ontology/parentCompany	http://dbpedia.org/resource/Marvel_Entertainment	"2"^^xsd:int	"0"^^xsd:int
4	http://dbpedia.org/resource/Marvel_Entertainment	http://dbpedia.org/ontology/keyPerson	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"0"^^xsd:int
5	http://dbpedia.org/resource/Guardians_of_the_Galaxy_(TV_series)	http://dbpedia.org/ontology/company	http://dbpedia.org/resource/Marvel_Television	"0"^^xsd:int	"1"^^xsd:int
6	http://dbpedia.org/resource/Marvel_Television	rdfs:seeAlso	http://dbpedia.org/resource/Marvel_Animation	"1"^^xsd:int	"1"^^xsd:int
7	http://dbpedia.org/resource/Marvel_Animation	http://dbpedia.org/ontology/parentCompany	http://dbpedia.org/resource/Marvel_Entertainment	"2"^^xsd:int	"1"^^xsd:int
8	http://dbpedia.org/resource/Marvel_Entertainment	http://dbpedia.org/ontology/keyPerson	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"1"^^xsd:int
9	http://dbpedia.org/resource/Guardians_of_the_Galaxy_(TV_series)	http://dbpedia.org/property/company	http://dbpedia.org/resource/Marvel_Television	"0"^^xsd:int	"2"^^xsd:int
10	http://dbpedia.org/resource/Marvel_Television	http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/controls	http://dbpedia.org/resource/Marvel_Animation	"1"^^xsd:int	"2"^^xsd:int
11	http://dbpedia.org/resource/Marvel_Animation	http://dbpedia.org/ontology/parentCompany	http://dbpedia.org/resource/Marvel_Entertainment	"2"^^xsd:int	"2"^^xsd:int
12	http://dbpedia.org/resource/Marvel_Entertainment	http://dbpedia.org/ontology/keyPerson	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"2"^^xsd:int
13	http://dbpedia.org/resource/Guardians_of_the_Galaxy_(TV_series)	http://dbpedia.org/ontology/company	http://dbpedia.org/resource/Marvel_Television	"0"^^xsd:int	"3"^^xsd:int
14	http://dbpedia.org/resource/Marvel_Television	http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/controls	http://dbpedia.org/resource/Marvel_Animation	"1"^^xsd:int	"3"^^xsd:int
15	http://dbpedia.org/resource/Marvel_Animation	http://dbpedia.org/ontology/parentCompany	http://dbpedia.org/resource/Marvel_Entertainment	"2"^^xsd:int	"3"^^xsd:int
16	http://dbpedia.org/resource/Marvel_Entertainment	http://dbpedia.org/ontology/keyPerson	http://dbpedia.org/resource/Stan_Lee	"3"^^xsd:int	"3"^^xsd:int

Which will be visualized like this:



All paths search with graph pattern — bound source and destination

Similarly to the example for shortest path search with graph pattern from earlier, we will be searching for all paths between source node **"The Black Panther (1977)"** and destination node **Stan Lee**, but this time excluding any properties of the type `http://dbpedia.org/property/keyPerson`. GraphDB will execute the path traversal using a graph pattern that specifies the exclusion of this property type through property negation with the SPARQL 1.1 property paths syntax.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>
```

(continues on next page)

(continued from previous page)

```
PREFIX dbp: <http://dbpedia.org/property/>

SELECT ?edge ?index ?path
WHERE {
    VALUES (?src ?dst) {
        ( dbr:The_Black_Panther_\(1977_film\) dbr:Stan_Lee )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:pathIndex ?path ;
                   path:resultBinding ?edge;
                   path:startNode ?start;
                   path:endNode ?end;
                   path:resultBindingIndex ?index .
        SERVICE <urn:path> {
            ?start !dbp:keyPerson ?end
        }
    }
}
```

Path edges are returned as RDF-star statements through the `path:resultBinding` property, and each edge is indexed with the `path:resultBindingIndex` property.

We can see that the first identified path excluding the DBpedia `keyPerson` property traverses the following nodes:

The movie "The Black Panther (1977)" – Marvel Studios – Marvel Entertainment – Stan Lee.

	edge	index	path
1	<< http://dbpedia.org/resource/The_Black_Panther_(1977_film) path:connectedTo http://dbpedia.org/resource/Black_Panther_(film) >>	"0"^^xsd:int	"0"^^xsd:int
2	<< http://dbpedia.org/resource/Black_Panther_(film) path:connectedTo http://dbpedia.org/resource/Marvel_Studios >>	"1"^^xsd:int	"0"^^xsd:int
3	<< http://dbpedia.org/resource/Marvel_Studios path:connectedTo http://dbpedia.org/resource/Marvel_Entertainment >>	"2"^^xsd:int	"0"^^xsd:int
4	<< http://dbpedia.org/resource/Marvel_Entertainment path:connectedTo http://dbpedia.org/resource/Stan_Lee >>	"3"^^xsd:int	"0"^^xsd:int

Note: Keep in mind that when using graph patterns we cannot view the predicates connecting the nodes. This is why GraphDB generates the predicate `http://www.ontotext.com/path#connectedTo` when exploring the path edges as RDF-star statements.

All paths search with N-ary relation

You might be familiar with the Six Degrees of Kevin Bacon parlor game where players arbitrarily choose an actor and then connect them to another actor via a film that both actors have starred in, repeating this process to try and find the shortest path that ultimately leads to famous US actor Kevin Bacon. The game is a reference to the six degrees of separation concept based on the assumption that any two people on Earth are six or fewer acquaintance links apart.

In this context, let's find all paths between source node **Chris Evans** and destination node **Chris Hemsworth** where the relationship between nodes is defined through an N-ary graph pattern based on actors co-starring in movies. The path search is limited with a minimum of two edges.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>


SELECT ?edge ?index ?path
WHERE {
    VALUES (?src ?dst) {
        ( dbr:Chris_Evans_\(actor\) dbr:Chris_Hemsworth )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:pathIndex ?path ;
                   path:minPathLength 2 ;
                   path:startNode ?start;
                   path:resultBinding ?edge ;
                   path:endNode ?end;
                   path:resultBindingIndex ?index .
        SERVICE <urn:path> {
            ?film a dbo:Film .
            ?film dbp:starring ?start .
            ?film dbp:starring ?end .
        }
    }
}
```

The following shows the first two returned paths:

	edge	index	path
1	<< http://dbpedia.org/resource/Chris_Evans_(actor) path:connectedTo http://dbpedia.org/resource/Mark_Ruffalo >>	"0"^^xsd:int	"0"^^xsd:int
2	<< http://dbpedia.org/resource/Mark_Ruffalo path:connectedTo http://dbpedia.org/resource/Chris_Hemsworth >>	"1"^^xsd:int	"0"^^xsd:int
3	<< http://dbpedia.org/resource/Chris_Evans_(actor) path:connectedTo http://dbpedia.org/resource/Cobie_Smulders >>	"0"^^xsd:int	"1"^^xsd:int
4	<< http://dbpedia.org/resource/Cobie_Smulders path:connectedTo http://dbpedia.org/resource/Mark_Ruffalo >>	"1"^^xsd:int	"1"^^xsd:int
5	<< http://dbpedia.org/resource/Mark_Ruffalo path:connectedTo http://dbpedia.org/resource/Chris_Hemsworth >>	"2"^^xsd:int	"1"^^xsd:int

3.11.3.3 Shortest distance

The following query finds the shortest distance between source node **Marvel Studios** and a date literal which represents Marvel Studios President **Kevin Feige's birthday**.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?dist
WHERE {
    VALUES (?src ?dst) {
        ( dbr:Marvel_Studios "1973-06-02"^^xsd:date )
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:distance ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:distanceBinding ?dist;
    }
}
```

We can see that the shortest path connecting them consists of two edges.

	dist
1	"2"^^xsd:int

3.11.3.4 Cyclic path

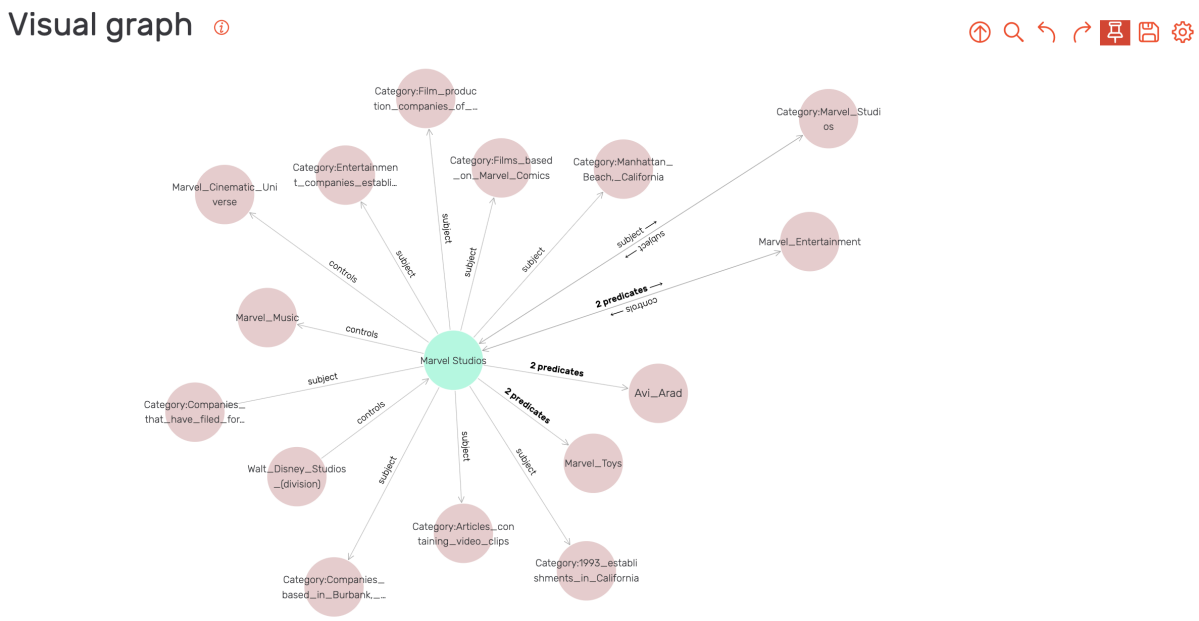
The following query finds **all paths that begin and end** with source node **Marvel Studios**.

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?edge ?index ?path
WHERE {
    VALUES (?src) {
        (dbr:Marvel_Studios)
    }
    SERVICE <http://www.ontotext.com/path#search> {
        <urn:path> path:findPath path:cycle ;
                   path:sourceNode ?src ;
                   path:resultBinding ?edge ;
                   path:pathIndex ?path ;
                   path:resultBindingIndex ?index .
    }
}
```

The following shows the first three returned paths:

	edge	index	path
1	<< http://dbpedia.org/resource/Marvel_Studios http://purl.org/dc/terms/subject http://dbpedia.org/resource/Category:Marvel_Studios >>	"0"^^xsd:int	"0"^^xsd:int
2	<< http://dbpedia.org/resource/Category:Marvel_Studios http://purl.org/dc/terms/subject http://dbpedia.org/resource/Marvel_Studios >>	"1"^^xsd:int	"0"^^xsd:int
3	<< http://dbpedia.org/resource/Marvel_Studios http://dbpedia.org/property/founder http://dbpedia.org/resource/Marvel_Entertainment >>	"0"^^xsd:int	"1"^^xsd:int
4	<< http://dbpedia.org/resource/Marvel_Entertainment http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/controls http://dbpedia.org/resource/Marvel_Studios >>	"1"^^xsd:int	"1"^^xsd:int
5	<< http://dbpedia.org/resource/Marvel_Studios http://dbpedia.org/ontology/foundedBy http://dbpedia.org/resource/Marvel_Entertainment >>	"0"^^xsd:int	"2"^^xsd:int
6	<< http://dbpedia.org/resource/Marvel_Entertainment http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/controls http://dbpedia.org/resource/Marvel_Studios	"1"^^xsd:int	"2"^^xsd:int

In the visual graph:



3.11.3.5 Parallel search mode

To demonstrate this, let's use the Shortest path search with a wildcard predicate example from earlier. To perform parallel path search, you need to set the `path:poolSize` property:

```
PREFIX path: <http://www.ontotext.com/path#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?pathIndex ?edgeIndex ?edge
WHERE {
    VALUES (?src ?dst) {
        ( dbr:The_Black_Panther_\(1977_film\) dbr:Stan_Lee )
    }
```

(continues on next page)

(continued from previous page)

```
    SERVICE path:search {
        [] path:findPath path:shortestPath ;
           path:sourceNode ?src ;
           path:destinationNode ?dst ;
           path:pathIndex ?pathIndex ;
           path:poolSize 8;
           path:resultBindingIndex ?edgeIndex ;
           path:resultBinding ?edge ;
           .
    }
}
```

The query will return the same results but execute faster.

3.11.3.6 Exportable graph pattern bindings

This query finds all paths between source node **Chris Evans** and destination node **Chris Hemsworth** where the relationship between nodes is defined through an N-ary graph pattern based on actors co-starring in movies. The path search is limited to a minimum of two edges. We also want to see the movies and their labels as part of the returned path.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX path: <http://www.ontotext.com/path#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?start ?end ?index ?path ?label ?film
WHERE {
    VALUES (?src ?dst) {
        ( dbr:Chris_Evans_\(actor\) dbr:Chris_Hemsworth )
    }
    SERVICE path:search {
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:pathIndex ?path ;
                   path:resultBindingIndex ?index ;
                   path:minPathLength 2 ;
                   path:startNode ?start;
                   path:exportBinding ?film ;
                   path:exportBinding ?label ;
                   path:endNode ?end;
                SERVICE <urn:path> {
                    ?film a dbo:Film .
                    ?film rdfs:label ?label .
                    ?film dbp:starring ?start .
                    ?film dbp:starring ?end .
                }
    }
}
```

Th following shows the first six returned paths:

	start	end	index	path	label	film
1	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Mark_Ruffalo	"0"^^xsd:int	"0"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
2	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"1"^^xsd:int	"0"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
3	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Mark_Ruffalo	"0"^^xsd:int	"1"^^xsd:int	"Avengers: Infinity War"@en	http://dbpedia.org/resource/Avengers:_Infinity_War
4	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"1"^^xsd:int	"1"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
5	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Cobie_Smulders	"0"^^xsd:int	"2"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
6	http://dbpedia.org/resource/Cobie_Smulders	http://dbpedia.org/resource/Mark_Ruffalo	"1"^^xsd:int	"2"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
7	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"2"^^xsd:int	"2"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
8	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Cobie_Smulders	"0"^^xsd:int	"3"^^xsd:int	"Avengers: Infinity War"@en	http://dbpedia.org/resource/Avengers:_Infinity_War
9	http://dbpedia.org/resource/Cobie_Smulders	http://dbpedia.org/resource/Mark_Ruffalo	"1"^^xsd:int	"3"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
10	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"2"^^xsd:int	"3"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
11	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Cobie_Smulders	"0"^^xsd:int	"4"^^xsd:int	"Captain America: The Winter Soldier"@en	http://dbpedia.org/resource/Captain_America:_The_W
12	http://dbpedia.org/resource/Cobie_Smulders	http://dbpedia.org/resource/Mark_Ruffalo	"1"^^xsd:int	"4"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
13	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"2"^^xsd:int	"4"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
14	http://dbpedia.org/resource/Chris_Evans_(act	http://dbpedia.org/resource/Robert_Downey_Jr	"0"^^xsd:int	"5"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
15	http://dbpedia.org/resource/Robert_Downey_	http://dbpedia.org/resource/Cobie_Smulders	"1"^^xsd:int	"5"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
16	http://dbpedia.org/resource/Cobie_Smulders	http://dbpedia.org/resource/Mark_Ruffalo	"2"^^xsd:int	"5"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron
17	http://dbpedia.org/resource/Mark_Ruffalo	http://dbpedia.org/resource/Chris_Hemsworth	"3"^^xsd:int	"5"^^xsd:int	"Avengers: Age of Ultron"@en	http://dbpedia.org/resource/Avengers:_Age_of_Ultron

Which in the visual graph would look like this:

3.11.3.7 Bidirectional search

This query finds the shortest bidirectional path between source node **The Black Panther** movie from 1977 and destination node **Marvel Studios**.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX path: <http://www.ontotext.com/path#>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?edge ?index ?path
WHERE {
    VALUES (?src ?dst) {
        ( dbr:The_Black_Panther_\(1977_film\) dbr:Marvel_Studios )
    }
    SERVICE path:search {
```

(continues on next page)

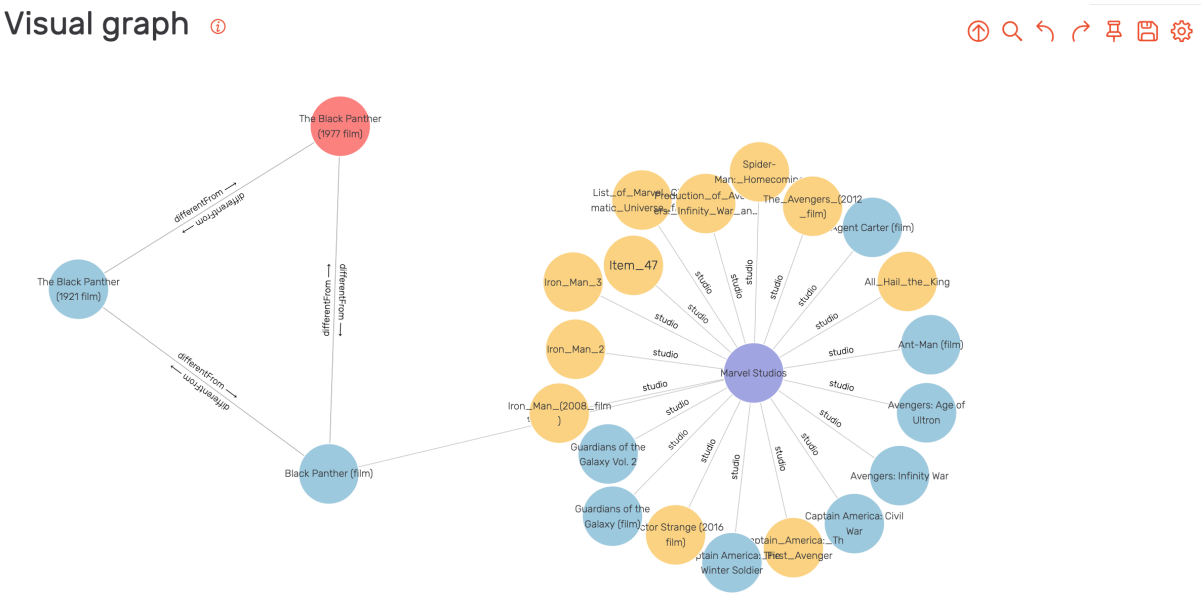
(continued from previous page)

```
        <urn:path> path:findPath path:allPaths ;
                   path:sourceNode ?src ;
                   path:destinationNode ?dst ;
                   path:resultBinding ?edge ;
                   path:pathIndex ?path ;
                   path:maxPathLength 4 ;
                   path:bidirectional true ;
                   path:resultBindingIndex ?index ;
    }
}
```

The following shows the first returned bidirectional path:

	edge	index	path
1	<< http://dbpedia.org/resource/The_Black_Panther_(1977_film) owl:differentFrom http://dbpedia.org/resource/Black_Panther_(film) >>	"0"^^xsd:int	"0"^^xsd:int
2	<< http://dbpedia.org/resource/Black_Panther_(film) http://dbpedia.org/property/producer http://dbpedia.org/resource/Kevin_Feige >>	"1"^^xsd:int	"0"^^xsd:int
3	<< http://dbpedia.org/resource/Kevin_Feige http://purl.org/dc/terms/subject http://dbpedia.org/resource/Category:Marvel_Studios >>	"2"^^xsd:int	"0"^^xsd:int
4	<< http://dbpedia.org/resource/Category:Marvel_Studios http://purl.org/dc/terms/subject http://dbpedia.org/resource/Marvel_Studios >>	"3"^^xsd:int	"0"^^xsd:int

In the visual graph:



3.12 Reasoning

Tip: To get the full benefit from this section, you need some basic knowledge of the two principle *Reasoning strategies* for rule-based inference: forward chaining and backward chaining.

GraphDB performs reasoning based on forward chaining of entailment rules defined using RDF triple patterns with variables. GraphDB's reasoning strategy is one of *Total materialization*, where the inference rules are applied repeatedly to the asserted (explicit) statements until no further inferred (implicit) statements are produced.

The GraphDB repository uses configured rulesets to compute all inferred statements at load time. To some extent, this process increases the processing cost and time taken to load a repository with a large amount of data. However, it has the desirable advantage that subsequent query evaluation can proceed extremely quickly.

3.12.1 Logical formalism

GraphDB uses a notation almost identical to R-Entailment defined by H.J.T. Horst. RDFS inference is achieved using a set of axiomatic triples and entailment rules. These rules allow the full set of valid inferences using RDFS semantics to be determined.

Horst defines RDFS extensions for more general rule support and a fragment of OWL, which is more expressive than Description Logic Programs and fully compatible with RDFS. First, he defines R-entailment, which extends RDFS-entailment in the following way:

- It can operate on the basis of any set of rules R — that is, it allows for extension or replacement of the standard set, defining the semantics of RDFS.
- It operates over so-called generalized RDF graphs, where blank nodes can appear as predicates (a possibility disallowed in RDF).
- Rules without premises are used to declare axiomatic statements.
- Rules without consequences are used to detect inconsistencies (integrity constraints).

Tip: To learn more, see *OWL compliance*.

3.12.2 Rule format and semantics

The rule format and the semantics enforced in GraphDB is analogous to R-entailment with the following differences:

- Free variables in the head (without binding in the body) are treated as blank nodes.

  Warning: Use this with extreme caution, because custom rulesets can easily be created that recursively infer an infinite number of statements making the semantics intractable.

- You can specify variable inequality constraints in addition to the triple patterns after after any premise or consequence. This leads to less complexity compared to R-entailment.
- The `cut` operator can be associated with rule premises. This optimization tells the rule compiler not to generate a variant of the rule with the identified rule premise as the first triple pattern.
- Context can be used for both rule premises and rule consequences allowing more expressive constructions that use ''intermediate'' statements contained within the given context URI.
- Consistency checking rules do not have consequences and will indicate an inconsistency when the premises are satisfied.

- Axiomatic triples can be provided as a set of statements, although these are not modeled as rules with empty bodies.

3.12.3 The ruleset file

GraphDB can be configured via rulesets — sets of axiomatic triples, consistency checks and entailment rules, which determine the applied semantics.

A ruleset file has three sections named `Prefices`, `Axioms`, and `Rules`. All sections are mandatory and must appear sequentially in this order. Comments are allowed anywhere and follow the Java convention of "`/* ... */`" for block comments and "`//`" for end of line comments.

For historical reasons, the way in which terms (variables, URLs and literals) are written differs from Turtle and SPARQL:

- URLs in prefix definitions are written without angle brackets
- variables are written without `?` or `$` and can include multiple alphanumeric characters
- URLs are written in brackets whether they use a prefix or are spelled in full
- datatype URLs are written without brackets, as shown here:

```
a <owl:maxQualifiedCardinality> "1"^^xsd:nonNegativeInteger
```

See the examples below and be careful when writing terms.

3.12.3.1 Prefixes

This section of the ruleset file defines the abbreviations for the namespaces used in the rest of the file. The syntax is:

```
shortname : URI
```

The following is an example of what a typical prefixes section might look like:

```
Prefices
{
    rdf  :  http://www.w3.org/1999/02/22-rdf-syntax-ns#
    rdfs :  http://www.w3.org/2000/01/rdf-schema#
    owl  :  http://www.w3.org/2002/07/owl#
    xsd  :  http://www.w3.org/2001/XMLSchema#
}
```

3.12.3.2 Axioms

This section of the ruleset file asserts axiomatic triples, which usually describe the meta-level primitives used for defining the schema such as `rdf:type` and `rdfs:Class`. It contains a list of the (variable free) triples, one per line.

For example, the RDF axiomatic triples are defined as shown here:

```
Axioms
{
    // RDF axiomatic triples
    <rdf:type>      <rdf:type> <rdf:Property>
    <rdf:subject>   <rdf:type> <rdf:Property>
    <rdf:predicate> <rdf:type> <rdf:Property>
    <rdf:object>    <rdf:type> <rdf:Property>
    <rdf:first>     <rdf:type> <rdf:Property>
    <rdf:rest>      <rdf:type> <rdf:Property>
```

(continues on next page)

(continued from previous page)

```
    <rdf:value>     <rdf:type> <rdf:Property>
    <rdf:nil>       <rdf:type> <rdf:List>
}
```

Note: Axiomatic statements are considered to be inferred for the purpose of query answering because they are a result of semantic interpretation defined by the chosen ruleset.

3.12.3.3 Rules

This section of the ruleset file is used to define entailment rules and consistency checks, which share a similar format. Each definition consists of premises and corollaries that are RDF statements, each defined with subject, predicate, object and optional context components. The subject and object can each be a variable, blank node, literal, a full URI, or the short name for a URI. The predicate can be a variable, a full URI, or a short name for a URI. If given, the context must be a full URI or a short name for a URI. Variables are alpha-numeric and must begin with a letter.

If the context is provided, the statements produced as rule consequences are not "visible" during normal query answering. Instead, they can only be used as input to this or other rules and only when the rule premise explicitly uses the given context as shown in the example below.

You can use inequality constraints to state that the values of the variables in a statement must not be equal to a specific full URI (or its short name), to a blank node, or to the value of another variable within the same rule. The behavior of an inequality constraint depends on whether it is placed in the body or the head of a rule. If it is placed in the body, then the whole rule will not "fire" if the constraint fails — that is, the constraint can be next to any statement pattern in the body of a rule with the same behavior. (The constraint does not have to be placed next to the variables it references.) If the constraint is in the head, then its location is significant because a constraint that does not hold will prevent only the statement it is adjacent to from being inferred.

Entailment rules

A rule definition has the following syntax:

```
Id: <rule_name>
    <premises> <optional_constraints>
    -------------------------------
    <consequences> <optional_constraints>
```

where each premise and consequence is on a separate line.

The following example illustrates some of the possibilities:

```
Rules
{
Id: rdf1_rdfs4a_4b
    x  a  y
    -------------------------------
    x  <rdf:type>  <rdfs:Resource>
    a  <rdf:type>  <rdfs:Resource>
    y  <rdf:type>  <rdfs:Resource>

Id: rdfs2
    x  a  y                 [Constraint a != <rdf:type>]
    a  <rdfs:domain>  z     [Constraint z != <rdfs:Resource>]
    -------------------------------
    x  <rdf:type>  z
```

(continues on next page)

(continued from previous page)

```
Id: owl_FunctProp
    p  <rdf:type>  <owl:FunctionalProperty>
    x  p  y       [Constraint y != z, p != <rdf:type>]
    x  p  z       [Constraint z != y] [Cut]
    ------------------------------
    y  <owl:sameAs>  z
}
```

The symbols `p`, `x`, `y`, `z` and `a` are variables. The second rule contains two constraints that reduce the number of bindings for each premise — they "filter out" those statements where the constraint does not hold.

In a forward chaining inference step, a rule is interpreted as meaning that for all possible ways of satisfying the premises, the bindings for the variables are used to populate the consequences of the rule. This generates new statements that will manifest themselves in the repository, for example, by being returned as query results.

The last rule contains an example of using the `Cut` operator, which is an optimization hint for the rule compiler. When rules are compiled, a different variant of the rule is created for each premise so that each premise occurs as the first triple pattern in one of the variants. This is done so that incoming statements can be efficiently matched to appropriate inferences rules. However, when a rule contains two or more premises that match identical triple patterns but use different variable names, the extra variant or variants are redundant and you can get better efficiency by simply not creating any extra rule variants.

In the above example, the rule `owl_FunctProp` would by default be compiled in three variants:

```
p  <rdf:type>  <owl:FunctionalProperty>
x  p  y
x  p  z
------------------------------
y  <owl:sameAs>  z


x  p  y
p  <rdf:type>  <owl:FunctionalProperty>
x  p  z
------------------------------
y  <owl:sameAs>  z


x  p  z
p  <rdf:type>  <owl:FunctionalProperty>
x  p  y
------------------------------
y  <owl:sameAs>  z
```

Here, the last two variants are identical apart from the rotation of variables `y` and `z`, so one of these variants is not needed. The use of the `Cut` operator above tells the rule compiler to eliminate this last variant: the one beginning with the premise `x p z`.

The use of context in rule bodies and rule heads is also best explained by an example. The following three rules implement the OWL2-RL property chain rule `prp-spo2`, and are inspired by the Rule Interchange Format (RIF) implementation:

```
Id: prp-spo2_1
    p <owl:propertyChainAxiom> pc
    start pc last                [Context <onto:_checkChain>]
    ---------------------------
    start p last

Id: prp-spo2_2
    pc <rdf:first> p
    pc <rdf:rest> t              [Constraint t != <rdf:nil>]
```

(continues on next page)

(continued from previous page)

```
    start p next
    next t last                 [Context <onto:_checkChain>]
    ----------------------------
    start pc last               [Context <onto:_checkChain>]

Id: prp-spo2_3
    pc <rdf:first> p
    pc <rdf:rest> <rdf:nil>
    start p last
    ----------------------------
    start pc last               [Context <onto:_checkChain>]
```

The RIF rules that implement `prp-spo2` use a relation (unrelated to the input or generated triples) called `_checkChain`. The GraphDB implementation maps this relation to the "invisible" context of the same name with the addition of `[Context <onto:_checkChain>]` to certain statement patterns. Generated statements with this context can only be used for bindings to rule premises when the exact same context is specified in the rule premise. The generated statements with this context will not be used for any other rules.

Inequality constraints in rules check whether a variable is bound to a blank node. If it is not, then the inference rule will fire:

```
Id: prp_dom

  a <rdfs:domain> b
  c a d
-------------------------------------
  c <rdf:type> b [Constraint p != blank_node]
```

Same-as optimization

The built-in OWL property `owl:sameAs` indicates that two URI references actually refer to the same thing. The following lines express the transitive and symmetric semantics of the rule:

```
/**
Id: owl_sameAsCopySubj
// Copy of statement over owl:sameAs on the subject. The support for owl:sameAs
// is implemented through replication of the statements where the equivalent
// resources appear as subject, predicate, or object. See also the couple of
// rules below
//
x <owl:sameAs> y [Constraint x != y]
x p z //Constraint p [Constrain p != <owl:sameAs>]
-------------------------------
y p z

Id: owl_sameAsCopyPred
// Copy of statement over owl:sameAs on the predicate
//
p <owl:sameAs> q [Constraint p != q]
x p y
-------------------------------
x q y

Id: owl_sameAsCopyObj
// Copy of statement over owl:sameAs on the object
//
x <owl:sameAs> y [Constraint x != y]
z p x //Constraint p [Constrain p != <owl:sameAs>]
-------------------------------
```

(continues on next page)

(continued from previous page)

```
z p y
**/
```

So, all nodes in the transitive and symmetric chain make relations to all other nodes. The relation coincides with the Cartesian NxN, hence the full closure contains N^2 statements. GraphDB optimizes the generation of excessive links by nominating an equivalence class representative to represent all resources in the symmetric and transitive chain. By default, the `owl:sameAs` optimization is enabled in all rulesets except when the ruleset is `empty`, `rdfs`, or `rdfsplus`. For additional information, check *Optimization of owl:sameAs*.

Consistency checks

Consistency checks are used to ensure that the data model is in a consistent state and are applied whenever an update transaction is committed. GraphDB supports consistency violation checks using standard OWL2-RL semantics. You can define rulesets that contain consistency rules. When creating a new repository, set the *check-for-inconsistencies configuration parameter* to `true`. It is `false` by default.

The syntax is similar to that of rules, except that `Consistency` replaces the `Id` tag that introduces normal rules. Also, consistency checks do not have any consequences and indicate an inconsistency whenever their premises can be satisfied, like in the following:

```
Consistency: something_can_not_be_nothing
    x rdf:type owl:Nothing
    ------------------------------


Consistency: both_sameAs_and_differentFrom_is_forbidden
    x owl:sameAs y
    x owl:differentFrom y
    ------------------------------
```

Consistency checks features

- Materialization and consistency mix: the rulesets support the definition of a mixture of materialization and consistency rules. This follows the existing naming syntax `id:` and `Consistency:`
- Multiple named rulesets: these are supported.
- No downtime deployment: The deployment of new and updated rulesets can be done to a running instance.
- Update transaction ruleset: Each update transaction can specify which named ruleset to apply. This is done by using “special” RDF statements within the update transaction.
- Consistency violation exceptions: if a consistency rule is violated, GraphDB throws exceptions. The exception includes details such as which rule has been violated and to which RDF statements.
- Consistency rollback: if a consistency rule is violated within an update transaction, the transaction will be rolled back and no statements will be committed.

In case of any consistency check failure, when a transaction is committed and consistency checking is switched on (by default it is off), then:

- A message is logged with details of what consistency checks failed
- An exception is thrown with the same details
- The whole transaction is rolled back

3.12.4 Rulesets

GraphDB offers several predefined semantics by way of standard rulesets (files), but can also be configured to use custom rulesets with semantics better tuned to the particular domain. The required semantics can be specified through the ruleset for each specific repository instance. Applications that do not need the complexity of the most expressive supported semantics can choose one of the less complex rulesets, which will result in faster inference.

Note: Each ruleset defines both rules and some schema statements, otherwise known as axiomatic triples. These read-only triples are inserted into the repository at initialization time and count towards the total number of reported "explicit" triples. The variation may be up to the order of hundreds depending upon the ruleset.

3.12.4.1 Predefined rulesets

The pre-defined rulesets provided with GraphDB cover various well-known knowledge representation formalisms and are layered in such a way that each extends the preceding one.

Ruleset	Description
empty	No reasoning. GraphDB operates as a plain RDF store.
rdfs	Supports the standard model-theoretic RDFS semantics. This includes support for `subClassOf` and related type inference as well as `subPropertyOf`.
rdfsplus	Extended version of RDFS with the additional support of symmetric, inverse and transitive properties, via the OWL vocabulary: `owl:SymmetricProperty`, `owl:inverseOf` and `owl:TransitiveProperty`.
owl-horst	OWL dialect close to OWL-Horst — essentially `pD*`.
owl-max	RDFS and that part of OWL Lite that can be captured in rules such as the deriving of functional and inverse functional properties, all-different, subclass by `union/enumeration`, and `min/max` cardinality constraints.
owl2-ql	The OWL2-QL profile — a fragment of OWL2 Full designed so that sound and complete query answering is LOGSPACE with respect to the size of the data. This OWL2 profile is based on DL-LiteR, a variant of DL-Lite that does not require the unique name assumption.
owl2-rl	The OWL2-RL profile — an expressive fragment of OWL2 Full that is amenable for implementation on rule engines.

Note: Not all rulesets support data-type reasoning, which is the main reason why OWL-Horst is not the same as `pD*`. The ruleset you need to use for a specific repository is defined through the *ruleset parameter*. There are optimized versions of all rulesets that avoid some little-used inferences.

Note: The default ruleset is RDFS-Plus (optimized).

OWL2-QL non-conformance

The implementation of OWL2-QL is non-conformant with the W3C OWL2 profiles recommendation as shown in the following table:

Conformant behavior	Implemented behavior
Given a list of disjoint (data or object) properties and an entity that is related with these properties to objects `{a, b, c, d,...}`, infer an `owl:AllDifferent` restriction on an anonymous list of these objects.	For each pair `{p, q}` `(p != q)` of disjoint (data or object) properties, infer the triple: `p owl:propertyDisjointWith q`, which is more likely to be useful for query answering.
For each class C in the knowledge base, infer the existence of an anonymous class that is the union of a list of classes containing only C.	Not supported. Even if this infinite expansion were possible in a forward chaining rule-based implementation, the resulting statements are of no use during query evaluation.
If `a` is an instance of C1 and `b` is an instance of C2, and C1 and C2 are disjoint, infer: `a owl:differentFrom b`	Impractical for knowledge bases with many members of pairs of disjoint classes such as Wordnet. Instead, this is implemented as a consistency check: If `x` is an instance of C1 and C2, and C1 and C2 are disjoint, then inconsistent.

3.12.4.2 Custom rulesets

GraphDB has an internal rule compiler that you can configure with a custom set of inference rules and axioms. You may define a custom ruleset in a `.pie` file (for example, `MySemantics.pie`). The easiest way to create a custom ruleset is to start modifying one of the `.pie` files that were used to build the precompiled rulesets.

Note: All pre-defined `.pie` files are included in the `configs/rules` folder of the *GraphDB distribution*.

If the code generation or compilation do not complete successfully, a Java exception is thrown indicating the problem. It will state either the `Id` of the rule, or the complete line from the source file that caused the problem. Line information is not preserved during the parsing of the rule file.

Specify the custom ruleset using the *ruleset configuration parameter*. There are optimized versions of all rulesets. The value of the `ruleset` parameter is interpreted as a filename and `.pie` is appended when not present. This file is processed to create Java source code that is compiled using the compiler from the Java Development Kit (JDK) version 1.6 or later.

Therefore, a prerequisite for using custom rulesets is that you use the Java Virtual Machine (JVM) from a JDK version 1.6 (or later) to run the application. If all goes well, GraphDB loads the class dynamically and instantiates it for further use during inference. The intermediate files are created in the folder named by the `java.io.tmpdir` system property. The JVM should have sufficient rights to read and write to this directory.

Note: Adding a custom ruleset to an existing repository is more complicated. You'll need to export or back up all explicit statements and recreate a new repository with the required ruleset. Once created, the explicit statements exported from the old repository can be imported to the new one.

3.12.5 Inference

Tip: See *Inference optimizations* for hints about performing inferencing more efficiently.

3.12.5.1 Compiling the reasoner

The GraphDB reasoner requires a `.pie` file of each ruleset to be compiled in order to instantiate. The process includes several steps:

1. Generate Java code out of the `.pie` file contents using the built-in GraphDB rule compiler.
2. Compile the Java code. This requires a JDK instead of JRE so that the Java compiler will be available through the standard Java instrumentation infrastructure.
3. Instantiate the Java code using a custom byte-code class loader.

Note: GraphDB supports dynamic extension of the reasoner with new rulesets.

3.12.5.2 Rulesets execution

- For each rule and each premise (triple pattern in the rule head), a `rule variant` is generated. We call this the "leading premise" of the variant. If a premise has the `Cut` annotation, no variant is generated for it.
- Every incoming triple (inserted or inferred) is checked against the leading premise of every rule variant. Since rules are compiled to Java bytecode on startup, this is very fast.
- If the leading premise matches, the rest of the premises are checked. This checking needs to access the repository, so it can be much slower.
  - GraphDB first checks premises with the least number of unbound variables.
  - For premises that have the same number of unbound variables, GraphDB follows the textual order in the rule.
- If all premises match, the conclusions of the rule are inferred.
- For each inferred statement:
  - If it does not exist in the default graph, it is stored in the repository and is queued for inference.
  - If it exists in the default graph, no duplicate statement is recorded. However, its "inferred" flag is still set. (see *How to manage explicit and implicit statements*).

3.12.5.3 Retraction of assertions

GraphDB stores explicit and implicit statements, the latter being the statements inferred (materialized) from the explicit statements. So, when explicit statements are removed from the repository, any implicit statements that rely on the removed statements must also be removed. To do this efficiently, GraphDB invalidates only the inferred statements that can no longer be derived in any way.

One approach is to maintain track information for every statement — typically the list of statements that can be inferred from this statement. The list is built up during inference as the rules are applied and the statements inferred by the rules are added to the lists of all statements that triggered the inferences. The drawback of this technique is that track information inflates more rapidly than the inferred closure — in the case of large datasets up to 90% of the storage is required just to store the track information.

Another approach is to perform backward chaining. Backward chaining does not require track information, since it essentially re-computes the tracks as required. Instead, a flag for each statement is used so that the algorithm can detect when a statement has been previously visited and thus avoid an infinite recursion.

GraphDB's algorithm follows these steps:

1. Apply a "visited" flag to all statements (`false` by default).
2. Store the statements to be deleted in the list L.

3. For each statement in L that is not visited yet, mark it as visited and apply the forward chaining rules. Statements marked as visited become invisible, which is why the statement must be first marked and then used for forward chaining.
4. If there are no more unvisited statements in L, then END.
5. Store all inferred statements in the list L1.
6. For each element in L1 check the following:
   - If the statement is a purely implicit statement (a statement can be both explicit and implicit and if so, then it is not considered purely implicit), mark it as deleted (prevent it from being returned by the iterators) and check whether it is supported by other statements. The `isSupported()` method uses queries that contain the premises of the rules, and the variables of the rules are preliminarily bound using the statement in question. That is to say, the `isSupported()` method starts from the projection of the query and then checks whether the query will return at least one result. In other words, this method performs backward chaining.
   - If a result is returned by any query (and every rule is represented by a query) in `isSupported()`, then this statement can still be derived from other statements in the repository, so it must not be deleted (its status is returned to "inferred").
   - If all queries return no results, then this statement can no longer be derived from any other statements, so its status remains "deleted" and the number of statements counter is updated.
7. L := L1 and execution resumes again at step 3.

Special care is taken when retracting `owl:sameAs` statements so that the algorithm still works correctly when modifying equivalence classes.

Note: One consequence of this algorithm is that deletion can still have poor performance when deleting schema statements, due to the (probably) large number of implicit statements inferred from them.

Note: The forward chaining part of the algorithm terminates as soon as it detects that a statement is read-only, because if it cannot be deleted, there is no need to look for statements derived from it. For this reason, performance can be greatly improved when all schema statements are made read-only by importing ontologies (and OWL and RDFS vocabularies) using the *imports repository parameter*.

Schema update transactions

When fast statement retraction is required, but you also need to update schemas, you can use a special statement pattern. Do this by including an insert for a statement with the following form in the update:

```
[] <http://www.ontotext.com/owlim/system#schemaTransaction> []
```

GraphDB will use the smooth-delete algorithm, but will also traverse read-only statements and allow them to be deleted and inserted. Such transactions are likely to be much more computationally expensive to achieve, so they are intended for the occasional, offline update to otherwise read-only schemas. The advantage is that fast-delete can still be used, but no repository export and import is required when making a modification to a schema.

For any transaction that includes an insert of the above special predicate/statement:

- Read-only (explicit or inferred) statements can be deleted
- New explicit statements are marked as read-only
- New inferred statements are marked:
  - Read-only if all the premises that fired the rule are read-only
  - Normal otherwise

Schema statements can be inserted or deleted using SPARQL UPDATE as follows:

```
DELETE {
  # [[schema statements to delete]]
}
INSERT {
  [] <http://www.ontotext.com/owlim/system#schemaTransaction> [] .
  # [[schema statements to insert]]
}
WHERE { }
```

3.12.6 Available ruleset operations

3.12.6.1 Operations on rulesets

All examples below use the `sys: namespace`, defined as:

```
prefix sys: <http://www.ontotext.com/owlim/system#>
```

Add a custom ruleset from a .pie file

The predicate `sys:addRuleset` adds a custom ruleset from the specified `.pie` file. The ruleset is named after the filename, without the `.pie` extension.

Example 1 This creates a new ruleset "test". If the absolute path to the file resides on, for example, `/opt/rules/test.pie`, it can be specified as `<file:/opt/rules/test.pie>`, `<file://opt/rules/test.pie>`, or `<file:///opt/rules/test.pie>`, i.e., with 1, 2, or 3 slashes. Relative paths are specified without the slashes or with a dot between the slashes: `<file:opt/rules/test.pie>`, `<file:/./opt/rules/test.pie>`, `<file://./opt/rules/test.pie>`, or even `<file:./opt/rules/test.pie>` (with a dot in front of the path). Relative paths can be used if you know the work directory of the Java process in which GraphDB runs.

```
INSERT DATA {
    _:b sys:addRuleset <file:c:/graphdb/test-data/test.pie>
}
```

Example 2 Same as above but creates a ruleset called "custom" out of the `test.pie` file found in the given absolute path.

```
INSERT DATA {
    <_:custom> sys:addRuleset <file:c:/graphdb/test-data/test.pie>
}
```

Example 3 Retrieves the `.pie` file from the given URL. Again, you can use `<_:custom>` to change the name of the ruleset to "custom" or as necessary.

```
INSERT DATA {
    _:b sys:addRuleset <http://example.com/test-data/test.pie>
}
```

Add a built-in ruleset

The predicate `sys:addRuleset` adds one of the built-in rulesets that GraphDB supports natively.

Example This adds the `"owl-max"` ruleset to the list of rulesets in the repository.

```
INSERT DATA {
    _:b sys:addRuleset "owl-max"
}
```

Add a custom ruleset with SPARQL INSERT

The predicate `sys:addRuleset` adds a custom ruleset from the specified `.pie` file. The ruleset is named after the filename, without the `.pie` extension.

Example This creates a new ruleset `"custom"`.

```
INSERT DATA {
    <_:custom> sys:addRuleset
        '''Prefices { a : http://a/ }
           Axioms {}
           Rules
           {
           Id: custom
              a b c
              a <a:custom1> c
              -----------------------
              b <a:custom1> a
           }'''
}
```

Note: Effects on the axiom set

When dealing with more than one ruleset, the result set of axioms is the UNION of all axioms of rulesets added so far. These are a special kind of statement that behave much like axioms in the sense that they can never be removed: `<P rdf:type rdf:Property>`, `<P rdfs:subPropertyOf P>`, `<X rdf:type rdfs:Resource>`.

These statements enter the repository just once — at the moment the property or resource is met for the first time. They remain in the repository forever, even if there are no more nodes related to that particular property or resource. (See *Rules optimizations* for more on this.)

List all rulesets

The predicate `sys:listRulesets` lists all rulesets available in the repository.

Example

```
SELECT ?state ?ruleset {
    ?state sys:listRulesets ?ruleset
}
```

Explore a ruleset

The predicate `sys:exploreRuleset` explores a ruleset.

Example

```
SELECT * {
    ?content sys:exploreRuleset "test"
}
```

Set a default ruleset

The predicate `sys:defaultRuleset` switches the default ruleset to the one specified in the object literal.

Example This sets the default ruleset to "test". All transactions use this ruleset, unless they specify another ruleset as a first operation in the transaction.

```
INSERT DATA {
    _:b sys:defaultRuleset "test"
}
```

Rename a ruleset

The predicate `sys:renameRuleset` renames the ruleset.

Example This renames the ruleset "custom" to "test". Note that "custom" is specified as the subject URI in the default namespace.

```
INSERT DATA {
    <_:custom> sys:renameRuleset "test"
}
```

Delete a ruleset

In the following, the predicate `sys:removeRuleset` deletes the ruleset `"test"` specified in the object literal.

Example

```
INSERT DATA {
    _:b sys:removeRuleset "test"
}
```

Note: Effects on the axiom set when removing a ruleset

When removing a ruleset, GraphDB just removes the mapping from the ruleset name to the corresponding inferencer. The axioms stay untouched.

Consistency check

The predicate `sys:consistencyCheckAgainstRuleset` checks if the repository is consistent with the specified ruleset.

Example

```
INSERT DATA {
    _:b sys:consistencyCheckAgainstRuleset "test"
}
```

3.12.6.2 Reinferring

Statements are inferred only when you insert new statements. So, if reconnecting to a repository with a different ruleset, it does not take effect immediately. You can trigger reinferencing with an update statement like the following:

```
INSERT DATA { [] <http://www.ontotext.com/owlim/system#reinfer> [] }
```

This removes all inferred statements and reinfers from scratch using the current ruleset. If a statement is both explicitly inserted and inferred, it is not removed. Statements of the type `<P rdf:type rdf:Property>`, `<P rdfs:subPropertyOf P>`, `<X rdf:type rdfs:Resource>`, and the axioms from all rulesets will stay untouched.

Tip: To learn more, see *How to manage explicit and implicit statements*.

3.12.7 Provenance

GraphDB's Provenance plugin enables the generation of inference closure from a specific named graph at query time. This is useful in situations where you want to trace what implicit statements are generated from a specific graph and the axiomatic triples part of the configured ruleset, that is, the ones inserted with a special predicate `sys:schemaTransaction`. Find out more about it in *Query time inference with the Provenance plugin*.

3.13 Exporting data

Data can be exported in several ways and formats.

3.13.1 JSON-LD and NDJSON-LD export forms

To export flattened, framed, or compacted document forms, you should also provide a context, a frame, or both as a file or a link. Use the `graphdb.jsonld.whitelist` *GraphDB configuration property* inside the `graphdb.properties` config file to provide a whitelist with URLs that are permitted for JSON-LD and NDJSON-LD processing.

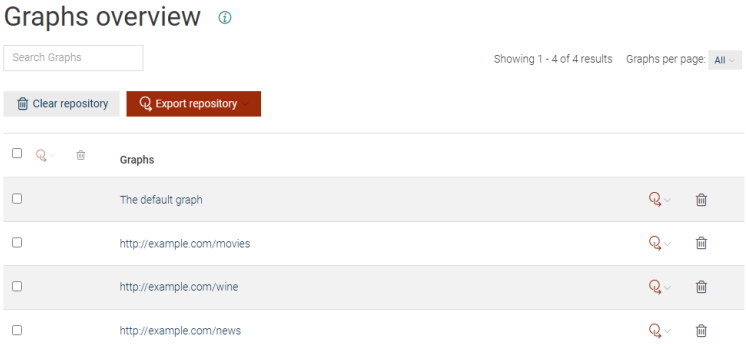
The whitelist is a comma-separated list of URLs. The wildcard (`*`) allows for fine-grained control, enabling administrators to specify a set of URLs, including entire directories. Each entry in the list represents a source that is considered safe for JSON-LD operations.

```
# Sets whitelist for JSON-LD and NDJSON-LD resources
graphdb.jsonld.whitelist = https://example.com/my/jsonld/*, file:///usr/local/my/jsonld/*
```

Note: A pop-up for selecting the form appears when exporting to JSON-LD or NDJSON-LD from Workbench. If you've selected flattened or compacted forms, a field for providing an address or file path to the context also appears. When a framed form is selected, a field for the path to the frame will appear instead.

3.13.2 Exporting a repository

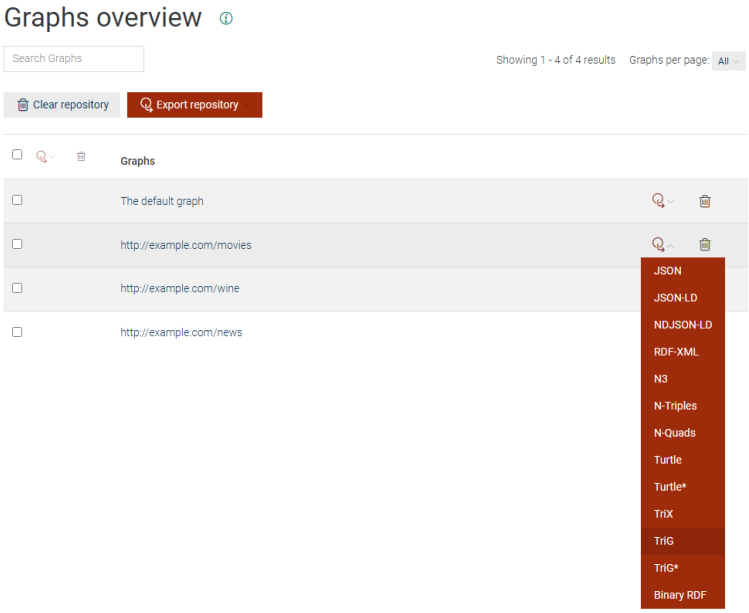
1. Go to *Explore* ‣ *Graphs overview*.
2. Click *Export repository* and then select the serialization to use for the exported file.



3.13.3 Exporting individual graphs

1. Go to *Explore* ‣ *Graphs overview*.
2. A list of contexts (graphs) in a repository is displayed. You can also search for particular graphs from the search field at the top of the screen.
3. After you click the checkboxes of one or more listed graphs, clicking *Export selected graphs* will prompt you to indicate a serialization format and then save a file in that format with the selected graphs.

To export a single graph, click the *Export graph* icon on the right side of its row on the display. To export multiple graphs, click their check boxes on the left to select them click the *Export selected graphs* icon at the top of the list.

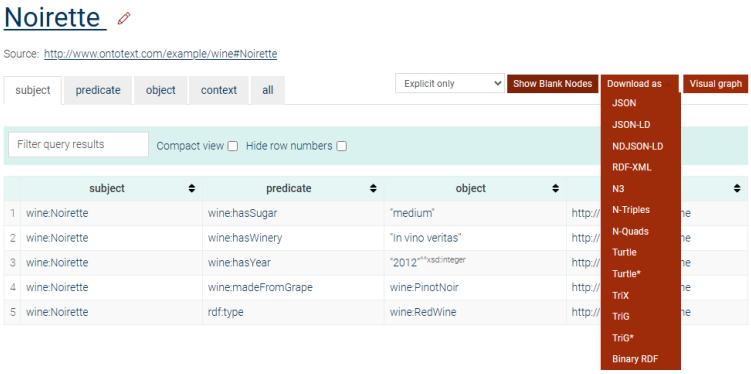


3.13.4 Exporting query results

You can export SPARQL query results from the SPARQL view by clicking the *Download As* button on the right side of the screen. The list of export formats will be RDF serializations after you execute a `CONSTRUCT` query or JSON, XML, CSV, and TSV formats after executing a `SELECT` query.

3.13.5 Exporting resources

After finding a resource from the *View resource* on GraphDB's home page, you can download its RDF triples in a format of your choice by clicking the *Download As* button on the right side of the screen:



3.13.6 Exporting via HTTP with curl

Using the `curl` command line utility lets you take advantage of GraphDB's *RDF4J REST API* to export repository data and queries from your operating system command line.

3.13.6.1 Request headers

- `Accept`: Relevant values for `GET` requests are the MIME types of *the supported RDF formats*.
- `Content-Type`: When exporting graph queries, used to specify the encoding of any request data sent to a server. Relevant values are the `MIME` types of *the supported RDF formats*.
- `Link`: Used only when exporting to JSON-LD or NDJSON-LD document form. When importing or exporting to compacted or flattened JSON-LD or NDJSON-LD document form, address or file path of context. When exporting to framed JSON-LD or NDJSON-LD document form, address or file path of frame that should be used by the Framing processor.

If a link is not provided when exporting to the JSON-LD or NDJSON serializations:

- **Exporting to compacted JSON-LD / NDJSON-LD document form** — exported data will use the namespaces of the repository instead.
- **Exporting to flattened JSON-LD / NDJSON-LD document form** — exported data will not contain context.
- **Exporting to framed JSON-LD / NDJSON-LD document form** — exported data will be exported to expanded JSON-LD or NDJSON-LD document form instead.

3.13.6.2 Fetch statements from repository

```
GET /repositories/<repo_id>/statements
```

Example:

```
curl '<base_url>/repositories/<repo_id>/statements' --header 'Accept: <MIME-type>/<MIME-subtype>'
```

Export JSON-LD using context example:

```
curl '<base_url>/repositories/<repo_id>/statements' \
--header 'Accept: application/ld+json; profile=http://www.w3.org/ns/json-ld#<type-of-mode>' \
--header 'Link: <link_to_the_context_location>; rel="http://www.w3.org/ns/json-ld#context"'
```

> **Warning:** The link to the context or frame location must be surrounded by angle brackets `<>`. From the example above, replace only `link_to_the_context_location` with your link.
>
> The following example would not work:
>
> ```
> --header 'Link: http://localhost:7200/json-ld-11/context.jsonld; rel="http://www.w3.org/ns/json-ld
> ↪#context"'
> ```
>
> Instead, it should be written like this:
>
> ```
> --header 'Link: <http://localhost:7200/json-ld-11/context.jsonld>; rel="http://www.w3.org/ns/json-ld
> ↪#context"'
> ```

Export JSON-LD using frame example:

```
curl '<base_url>/repositories/<repo_id>/statements' \
--header 'Accept: application/ld+json;profile=http://www.w3.org/ns/json-ld#framed' \
--header 'Link: <link_to_the_frame_location>; rel="http://www.w3.org/ns/json-ld#frame"'
```

3.13.6.3 Fetch queries from repository

```
GET /repositories/<repo_id>/
```

The query can be specified with `--data-raw`. It can also be provided with `--data-urlencode query@/<path-to-query-file>`

Example:

```
curl '<base_url>/repositories/<repo_id>' \
--header 'Content-Type: application/x-www-form-urlencoded; charset=UTF-8' \
--header 'Accept: application/<MIME-type>' \
--data-raw $'query=<sparql-query>'
```

Export JSON-LD example:

```
curl '<base_url>/repositories/<repo_id>' \
--header 'Content-Type: application/x-www-form-urlencoded; charset=UTF-8' \
--header 'Accept: application/ld+json; profile=http://www.w3.org/ns/json-ld#<type-of-mode>' \
--header 'Link: <link_to_the_context_location>; rel="http://www.w3.org/ns/json-ld#context"' \
--data-raw $'query=<sparql-query>'
```

> **Warning:** The link to the context or frame location must be surrounded by angle brackets `<>`. From the example above, replace only `link_to_the_context_location` with your link.

The following example would not work:

```
--header 'Link: http://localhost:7200/json-ld-11/context.jsonld; rel="http://www.w3.org/ns/json-ld
↪#context"'
```

Instead, it should be written like this:

```
--header 'Link: <http://localhost:7200/json-ld-11/context.jsonld>; rel="http://www.w3.org/ns/json-ld
↪#context"'
```

3.13.7 Configuring the JSON-LD writer properties

The JSON-LD writer has additional properties that configure how the writer operates:

Property name	Description	Default value
`com.ontotext.graphdb.rio.jsonld_processing_mode`	Defines whether the writer will use JSON-LD 1.0 or JSON-LD 1.1 specifications when processing JSON-LD exports.	`V1_1`
`com.ontotext.graphdb.rio.jsonld.frame`	Provides a path to a frame that is used when exporting to framed document form.	`null`
`com.ontotext.graphdb.rio.jsonld.compact_to_relative`	Determines if IRIs are compacted relative to the base option or document location when compacting.	`TRUE`
`com.ontotext.graphdb.rio.jsonld.expand_context`	Provides a path to a context that is used to expand the active context when expanding a document.	`null`
`com.ontotext.graphdb.rio.jsonld.ordered`	If set to `TRUE`, certain algorithm processing steps are ordered lexicographically where indicated. If set to `FALSE`, order is not considered during processing.	`FALSE`
`com.ontotext.graphdb.rio.jsonld.explicit`	Indicates whether the output should be in explicit form. When set to `TRUE`, the output will include redundant information to improve human readability.	`FALSE`
`com.ontotext.graphdb.rio.jsonld_embed`	When set to `ALWAYS`, node objects are embedded as property values, unless this would cause a circular reference. When set to `NEVER`, node references are used when serializing matching values. When set to `ONCE`, only a single value within a given node object will be embedded, and other values of other properties will use a node reference.	`ONCE`
`com.ontotext.graphdb.rio.jsonld.omit_default`	Controls whether default values are omitted from the output.	`FALSE`
`com.ontotext.graphdb.rio.jsonld.omit_graph`	Controls whether the top-level graph is omitted from the output.	`TRUE`
`com.ontotext.graphdb.rio.jsonld.require_all`	Specifies whether all properties are considered as required.	`FALSE`
`com.ontotext.graphdb.rio.jsonld.numeric_id`	Allows the use of numeric values as `@id` in the exported JSON-LD document.	`FALSE`
`com.ontotext.graphdb.rio.jsonld.url_validation`	Specifies whether URI validation is enabled. When set to `TRUE`, URI validation is performed during JSON-LD processing.	`TRUE`

CHAPTER

FOUR

CONNECTORS

4.1 Connecting to external components and services

The GraphDB Connectors enable connections with external components and services. These let you use the features of those components and services with the data stored in your GraphDB repositories, providing new possibilities for what your applications can do with that data. Connectors offer the additional benefit of staying automatically up to date with the GraphDB repository data.

Full-text search tools perform single or multiple word linguistic searches against a unit of searching called a "document". In the context of GraphDB, a "document" is defined as an entity with a unique identifier (*URI*) and a set of property names and values. In an *RDF* context, this corresponds to a set of triples that have the same subject — typically, an instance of some type. Documentation for these text search tools describe how they index and search these "documents".

When you use a full-text search connector with GraphDB, you can embed queries from that tool in a SPARQL query. These queries can take advantage of that tool's own specialized text search and aggregation operators to perform very fast searches for patterns that standard SPARQL cannot find. The connected tool makes this possible by creating an index that it keeps updated as the data in the GraphDB repository evolves.

The Connectors also support property chains — sequences of triples where each triple's object is the subject of the subsequent triple, which enables easier navigation through a knowledge graph to retrieve data values. Property chains are specified on the dialog box that you use to configure a new connector. A property chain is useful, for example, if you want to synchronize a department head name field in the external system with a chain of RDF triples leading from a department URI to a department manager URI to the manager's name property.

4.1.1 Full-text search and aggregation connectors

GraphDB comes with the following full-text search (FTS) connector implementations:

- *Lucene GraphDB Connector*
- *Solr GraphDB Connector* (requires a **GraphDB Enterprise** license)
- *Elasticsearch GraphDB Connector* (requires a **GraphDB Enterprise** license)
- *OpenSearch GraphDB Connector* (requires a **GraphDB Enterprise** license)

Table 1: *Features Comparison Table: FTS and Aggregation Connectors*

Feature	Lucene	Solr	Elasticsearch	OpenSearch
FTS search	✓	✓	✓	✓
Simple facets	✓	✓	✓	✓
Sorting	✓	✓	✓	✓
Snippet extraction	✓	✓	✓	✓
Limit and offset	✓	✓	✓	✓
Fixed range facets	×	✓	✓	✓
Variable range facets	×	✓	✓	✓
Nested facets	×	✓	✓	✓
Aggregations • terms • histogram • range • min/max • sum • average • count • standard deviation • variance • sum of squares	×	×	✓	✓
Sub-aggregations	×	×	✓	✓

4.1.2 ChatGPT Retrieval connector

The *ChatGPT Retrieval GraphDB Connector* lets you convert an RDF model to a text representation and synchronize it to the ChatGPT Retrieval Plugin, which converts the text document to embedded vectors and index it into its configured vector database. This is an experimental feature that is mostly meant to be used together with the *Talk to Your Graph* feature but can also be used independently to query the vector database.

4.1.3 MongoDB integration

The *MongoDB integration* lets you use SPARQL to query MongoDB databases with heterogeneous joins of that data to RDF data. A document-based database with a huge community of developers and users, MongoDB is part of the MEAN technology stack and guarantees scalability and performance well beyond the throughput supported in GraphDB. The integration between GraphDB and MongoDB is done by a plugin that sends a request to MongoDB and then transforms the result to an RDF model.

4.1.4 Kafka connector

Note: Despite having a similar name, the *Kafka Sink connector* is not a GraphDB connector.

The *Kafka GraphDB Connector* lets you synchronize changes in the RDF model to any downstream system via the Kafka framework. This enables easy processing of RDF updates in any external system and covers a variety of use cases where you need a reliable synchronization mechanism.

Using the Kafka connector requires a **GraphDB Enterprise** license.

4.2 General full-text search with the Connectors

The GraphDB Connectors offer an excellent solution for indexing data with a well-known schema — for example, index documents that have type A, where each document has a field F1 that can be reached by following the property chain composed of property IRIs P1 and P2.

The features described below add a more general full-text search (FTS) ability to the connectors. You can use them individually or combined as desired to meet the specific needs of the use case.

Note: See more about GraphDB's FTS capabilities in the *Full-text search* section of this documentation.

The connector features described below are useful when defining a connector for general full-text search.

4.2.1 Indexing with wildcard literals

This feature lets you index literals without specifying the IRI of the predicate that leads to the literal. Use `$literal` as the last element of the property chain.

See more about wildcard literals in the *Lucene connector*, the *Solr connector*, the *Elasticsearch connector*, and the *OpenSearch connector* sections.

4.2.2 Deriving field names from a predicate URI

This feature lets you use dynamic field names derived from the IRI of the last predicate in the property chain.

See more about field name transformations in the *Lucene connector*, the *Solr connector*, the *Elasticsearch connector*, and the *OpenSearch connector* sections.

4.2.3 Any type or untyped indexing

Specify `$any` or `$untyped` as the sole type to index all entities that have at least one RDF type, or all entities regardless of whether they have any RDF type.

See more about types in the *Lucene connector*, the *Solr connector*, the *Elasticsearch connector*, and the *OpenSearch connector* sections.

4.2.4 Examples

All examples use the Star Wars RDF dataset. Download `starwars-data.ttl` and import it into a fresh repository before trying any of these examples.

The example connector definitions use the Lucene connector but can be easily adapted to Solr, Elasticsearch and OpenSearch by changing `lucene` in the prefix definitions to `solr`, `elasticsearch` or `opensearch`, and adding any additional parameters required by the respective connector such as `elasticsearchNode`.

4.2.4.1 Indexing all literals

To index all literals in the repository regardless of where they are attached in the graph, you can combine *wildcard literal indexing* and *untyped indexing*. Create a connector with an update request like the following:

```
PREFIX con: <http://www.ontotext.com/connectors/lucene#>
PREFIX con-inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
    con-inst:starwars_fts con:createConnector '''
    {
      "fields": [
        {
          "fieldName": "fts",
          "propertyChain": [
            "$literal"
          ],
          "facet": false
        }
      ],
      "languages": [
        ""
      ],
      "types": [
        "$untyped"
      ]
    }
''' .
}
```

The connector defines a single field, `fts`, that will index all literals regardless of their predicate: `$literal` as the last element of the property chain. The connector has no type expectations on the entities that lead to those literals and will index any entity regardless of whether it has an RDF type: `$untyped` in the `types` parameter.

Since the Star Wars dataset contains literals in many different languages, we restrict the index definition further by specifying `""` (the empty language, which means any literal without a language tag) using the `languages` option.

We can now search in this connector as usual, for example for the FTS query “luke skywalker”:

```
# Full-text search for "skywalker"
PREFIX con: <http://www.ontotext.com/connectors/lucene#>
PREFIX con-inst: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?entity ?label {
    [] a con-inst:starwars_fts ;
       con:query "skywalker" ;
       con:entities ?entity .
    ?entity rdfs:label ?label
    FILTER(lang(?label) = "")
}
```

We get many different results belonging to different types; below are the first ten results:

Table 2: SPARQL results for "skywalker"

?entity	?label
<https://swapi.co/resource/human/43>	Shmi Skywalker
<https://swapi.co/resource/human/1>	Luke Skywalker
<https://swapi.co/resource/human/35>	Padmé Amidala
<https://swapi.co/resource/planet/1>	Tatooine
<https://swapi.co/resource/human/10>	Obi-Wan Kenobi
<https://swapi.co/resource/human/11>	Anakin Skywalker
<https://swapi.co/resource/droid/2>	C-3PO
<https://swapi.co/resource/human/4>	Darth Vader
<https://swapi.co/resource/droid/3>	R2-D2
<https://swapi.co/resource/human/18>	Wedge Antilles

4.2.4.2 Indexing all literals in distinct fields

The above example indexes all literals into a single field, which is convenient for very rough full-text search. You can fine-tune it by using *field names derived from the predicate*. In this example, we added `"fieldNameTransform": "predicate.localName"` so we will get a field for every predicate whose object literal is indexed, and the field name will be derived from the local name of the predicate:

```
PREFIX con: <http://www.ontotext.com/connectors/lucene#>
PREFIX con-inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
    con-inst:starwars_fts2 con:createConnector '''
    {
      "fields": [
        {
          "fieldName": "fts",
          "fieldNameTransform": "predicate.localName",
          "propertyChain": [
            "$literal"
          ],
          "facet": false
        }
      ],
      "languages": [
        ""
      ],
      "types": [
        "$untyped"
      ]
    }
''' .
}
```

We can use this connector to do general full-text searches, but also more precise ones, such as a query only of entity labels (the field `label` is the result of taking the local name of `<http://www.w3.org/2000/01/rdf-schema#label>` at indexing time):

```
# Full-text search for "skywalker" in the field "label"
PREFIX con: <http://www.ontotext.com/connectors/lucene#>
PREFIX con-inst: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?entity ?label {
    [] a con-inst:starwars_fts2 ;
       con:query "label:skywalker" ;
```

(continues on next page)

(continued from previous page)

```
        con:entities ?entity .
    ?entity rdfs:label ?label
    FILTER(lang(?label) = "")
}
```

We get only three results back — the people that have "Skywalker" in their name:

Table 3: SPARQL results for "skywalker" in the field "label"

?entity	?label
`<https://swapi.co/resource/human/43>`	Shmi Skywalker
`<https://swapi.co/resource/human/1>`	Luke Skywalker
`<https://swapi.co/resource/human/11>`	Anakin Skywalker

4.3 Lucene GraphDB Connector

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Lucene but with the additional benefit of staying automatically up to date with the GraphDB repository data.

Note: GraphDB supports *full-text search* options as well.

The Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

The main features of the GraphDB Connectors are:

- maintenance of an index that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entity synchronization defined by:
  - a list of fields (on the Lucene side) and property chains (on the GraphDB side) whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization (the default is all languages)
  - additional filtering by property and value
- full-text search using native Lucene queries
- snippet extraction: highlighting of search terms in the search result
- faceted search
- sorting by any preconfigured field
- paging of results using `offset` and `limit`
- custom mapping of RDF types to Lucene types
- specifying which Lucene analyzer to use (the default being Lucene's `StandardAnalyzer`)
- optionally stripping HTML and XML tags in literals
- boosting an entity by the numeric value of one or more predicates
- custom scoring expressions at query time to evaluate a total score based on Lucene score and entity boost

Each feature is described in detail below.

4.3.1 Usage

All interactions with the Lucene GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the Lucene GraphDB Connector, this is `http://www.ontotext.com/connectors/lucene#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/lucene#createConnector`) to create a connector instance for Lucene.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For Lucene, the instance IRI is `http://www.ontotext.com/connectors/lucene/instance#`.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito and Rozova as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
```

(continues on next page)

(continued from previous page)

```
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

4.3.2 Setup and maintenance

Third-party component versions This version of the Lucene GraphDB Connector uses Lucene version 9.12.0.

4.3.2.1 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- classes to synchronize
- properties to synchronize

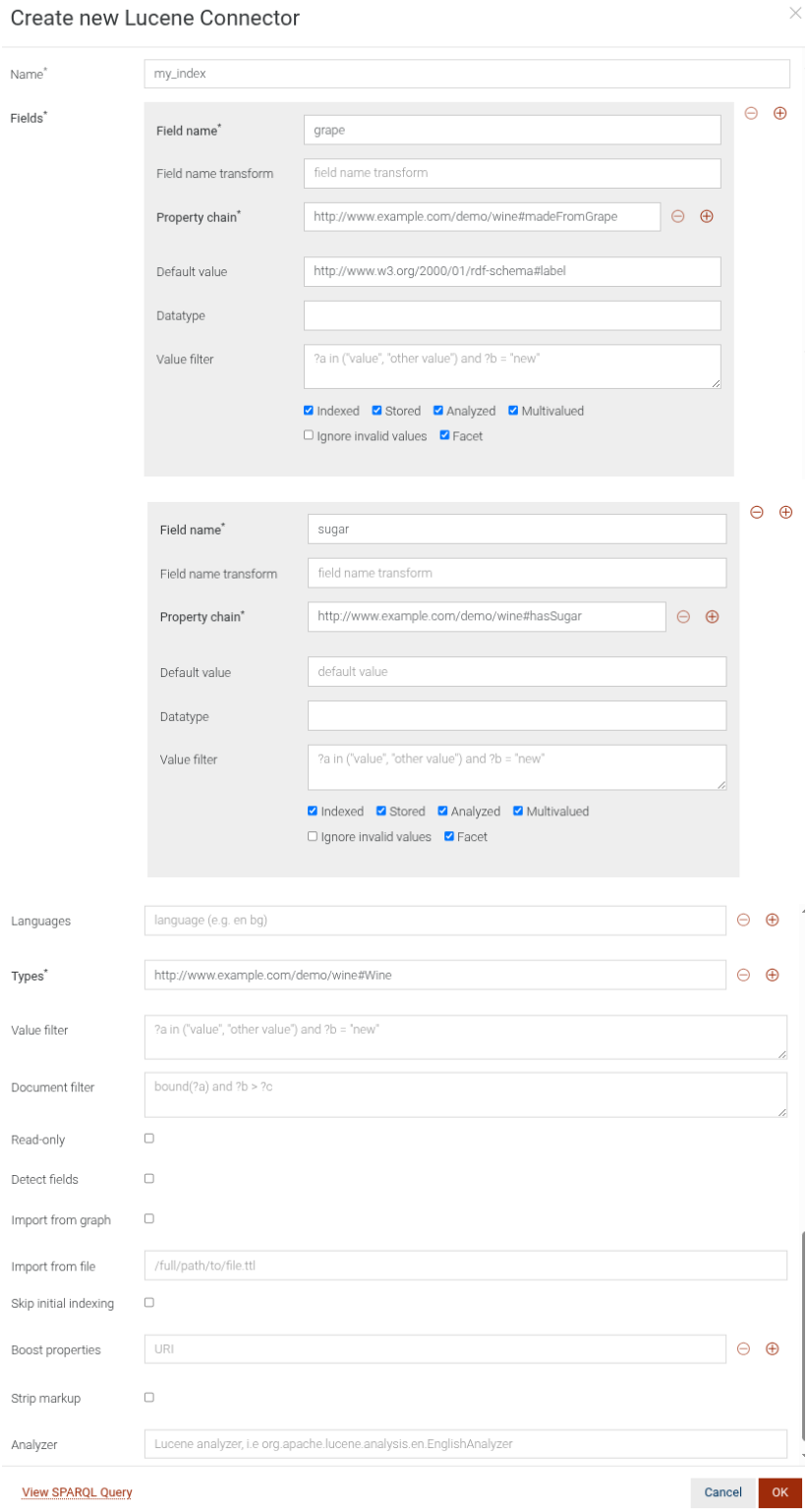
The configuration data must be provided as a JSON string representation and passed together with the create command.

You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill out the configuration form.



4. Execute the `CREATE` statement from the form by clicking *OK*. Alternatively, you can view its SPARQL query by clicking *View SPARQL Query*, and then copy it to execute it manually or integrate it in automation scripts.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `luc:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
    luc-index:my_index luc:createConnector '''
{
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false,
      "multivalued": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ]
}
''' .
}
```

The above command creates a new Lucene connector instance.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to Lucene. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: "grape", "sugar", and "year". The field names are used later in the queries.

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine's `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to the wine.

The fields `sugar` and `year` contain discrete values such as *medium*, *dry*, *2012*, and *2013*, so it is best to specify the option `analyzed: false` as well. See `analyzed` in *Defining fields* for more information.

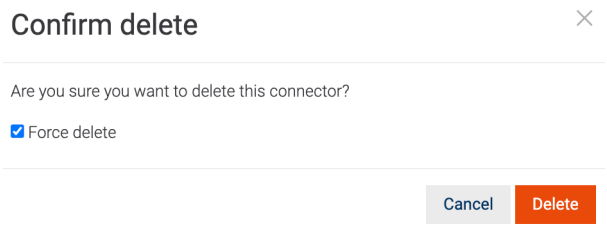
4.3.2.2 Dropping a connector instance

Dropping (deleting) a connector instance removes all references to its external store from GraphDB as well as all Lucene files associated with it.

The drop command is triggered by a SPARQL `INSERT` with the `luc:dropConnector` predicate where the name of the connector instance must be in the subject position. For example, the following removes the connector `my_index`:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon 🗑, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.3.2.3 Retrieving the create options for a connector instance

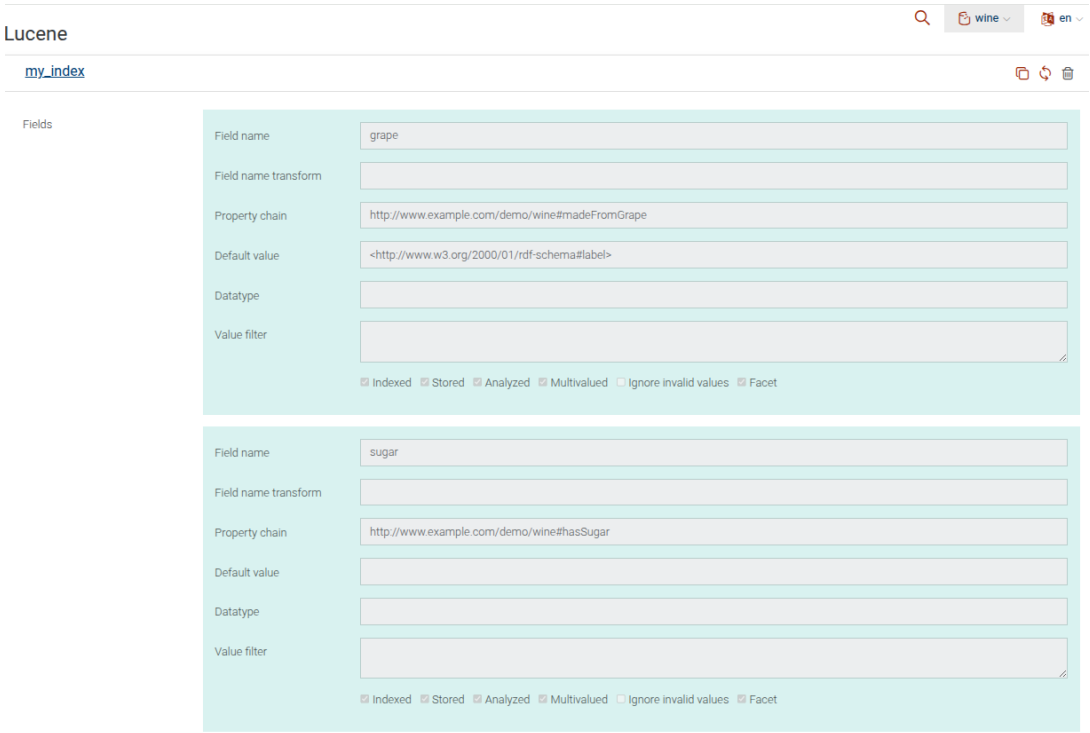
You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?createString {
  luc-index:my_index luc:listOptionValues ?createString .
}
```

4.3.2.4 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* 🔄 or *Delete* 🗑 icons to perform these operations. Click the *Copy* 📋 icon to copy the connector definition query to your clipboard.



With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

SELECT ?cntUri ?cntStr {
  ?cntUri luc:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/lucene/instance#my_index>`, while `?cntStr` is bound to a string representing the local name such as “my_index” in this example.

4.3.2.5 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>

SELECT ?cntUri ?cntStatus {
  ?cntUri luc:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.3.3 Working with data

4.3.3.1 Adding, updating, and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` update requests. This is achieved by intercepting all changes in the plugin and determining which Lucene documents need to be updated.

4.3.3.2 Simple queries

Once a connector instance has been created, you can use SPARQL to query data from it. For each matching Lucene document, the connector instance returns the document subject. In its simplest form, you query that data using a `SELECT` query that includes a Lucene query as the object of the `luc:query` predicate:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity {
  ?search a luc-index:my_index ;
      luc:query "grape:cabernet" ;
      luc:entities ?entity .
}
```

The result of this query binds `?entity` to the two wines made from grapes that have "cabernet" in their name: `:Yoyowine` and `:Franvino`.

Note: You must use the field names you chose when you created the connector instance. They can be identical to the property IRIs but you must escape any special characters according to what Lucene expects.

The creation of such a query follows these basic steps:

1. Get a query instance of the requested connector instance by using the RDF notation `"X a Y" (= X rdf:type Y)`, where `X` is a variable and `Y` is a connector instance IRI. `X` is bound to a query instance of the connector instance.
2. Assign a query to the query instance by using the system predicate `luc:query`.
3. Request the matching entities through the `luc:entities` predicate.

You can also provide per-query search options by using one or more option predicates. The option predicates are described in detail below.

4.3.3.3 Combining Lucene results with GraphDB data

You can use the bound `?entity` variable in other SPARQL triple patterns to build complex queries that join to or fetch additional data from GraphDB. For example, to see the actual grapes in the matching wines as well as the year they were made:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?entity ?grape ?year {
  ?search a luc-index:my_index ;
      luc:query "grape:cabernet" ;
      luc:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
```

(continues on next page)

(continued from previous page)

```
  ?entity wine:hasYear ?year
}
```

The result looks like this:

	entity	grape	year
1	wine:Yoyowine	wine:CabernetSauvignon	"2013"^^xsd:integer
2	wine:Franvino	wine:Merlot	"2012"^^xsd:integer
3	wine:Franvino	wine:CabernetFranc	"2012"^^xsd:integer

Note: `:Franvino` is returned twice because it is made from two different grapes, both of which are returned.

4.3.3.4 Entity match score

It is possible to access the match score returned by Lucene with the `score` predicate. As each entity has its own score, the predicate should come at the entity level. For example:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity ?score {
  ?search a luc-index:my_index ;
      luc:query "grape:cabernet" ;
      luc:entities ?entity .
  ?entity luc:score ?score
}
```

The result looks like this but the actual score might be different as it depends on the specific Lucene version:

	entity	score
1	wine:Yoyowine	"0.3979403078556061"^^xsd:double
2	wine:Franvino	"0.33036553859710693"^^xsd:double

4.3.3.5 Basic facet queries

You can also query facets with your connector instance. For example, with the sample `my_index` connector described above:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?facetName ?facetValue ?facetCount WHERE {
  # Note empty query is allowed and will just match all documents, hence no :query
  ?r a luc-index:my_index ;
    luc:facetFields "year,sugar" ;
    luc:facets [
     luc:facetName ?facetName;
     luc:facetValue ?facetValue;
     luc:facetCount ?facetCount
   ]
}
```

It is important to specify the facet fields by using the `facetFields` predicate. Its value is a simple comma-delimited list of field names. In order to get the faceted results, use the `luc:facets` predicate. Because each facet has three components (name, value and count), the `luc:facets` predicate returns multiple nodes that can be used to access the individual values for each component through the predicates `facetName`, `facetValue`, and `facetCount`.

The resulting bindings look like the following:

	facetName	facetValue	facetCount
1	"year"	"2012"	"3"^^xsd:long
2	"year"	"2013"	"2"^^xsd:long
3	"sugar"	"dry"	"3"^^xsd:long
4	"sugar"	"medium"	"2"^^xsd:long

You can easily see that there are three wines produced in 2012 and two in 2013. You also see that three of the wines are dry, while two are medium. However, it is not necessarily true that the three wines produced in 2012 are the same as the three dry wines as each facet is computed independently.

4.3.3.6 Sorting

You can sort the entities returned by a connector query according to one or more fields. To do this, assign a comma-delimited list of sort fields to the `orderBy` predicate. Each field can be prefixed with a minus to indicate sorting in descending order. For example:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>
SELECT ?entity ?sugar{
  ?search a luc-index:my_index ;
      luc:query "year:2013" ;
      luc:orderBy "-sugar" ;
      luc:entities ?entity.
  ?entity wine:hasSugar ?sugar
}
```

The result of this sample query contains wines produced in 2013 sorted according to their sugar content in descending order:

	entity	sugar
1	wine:Rozova	"medium"
2	wine:Yoyowine	"dry"

By default, entities are sorted according to their matching score in descending order.

Note: If you join the entity from the connector query to other triples stored in GraphDB, GraphDB might scramble the order. Avoid this by using SPARQL's `ORDER BY` instruction.

Tip: Sorting by an analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens and sorting uses the least (in the lexicographical sense) token. For example, "North America" will be sorted before "Europe" because the token "america" is lexicographically smaller than the token "europe". If you need to sort by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.3.3.7 Limit and offset

Limit and offset are supported on the Lucene side of the query. You can set these values using the predicates `limit` and `offset`. Consider this example in which an offset of `1` and a limit of `1` are specified:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity {
  ?search a luc-index:my_index ;
      luc:query "sugar:dry" ;
      luc:offset "1" ;
      luc:limit "1" ;
      luc:entities ?entity .
}
```

`offset` is counted from 0. The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:

	entity
1	wine:Yoyowine
2	wine:Franvino
3	wine:Blanquito

Note: The specific order in which GraphDB returns the results depends on how Lucene returns the matches, unless sorting is specified.

4.3.3.8 Snippet extraction

Snippet extraction lets you retrieve highlighted snippets of text that match a query. You can access the snippets by using the the dedicated predicate `luc:snippets`. This binds a blank node that in turn provides the actual snippets via the predicates `luc:snippetField` and `luc:snippetText`. The predicate snippets must be attached to the entity, as each entity has a different set of snippets. For example, in a search for Cabernet:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a luc-index:my_index ;
      luc:query "grape:cabernet" ;
      luc:entities ?entity .
  ?entity luc:snippets ?snippet .
  ?snippet luc:snippetField ?snippetField ;
     luc:snippetText ?snippetText .
}
```

the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

	entity	snippetField	snippetText
1	wine:Yoyowine	"grape"	"Cabernet Sauvignon"
2	wine:Franvino	"grape"	"MerlotCabernet Franc"

Note: The actual snippets might be different depending on the specific Lucene implementation.

You can tweak how the snippets are collected and composed by using the following option predicates:

- `luc:snippetSize`: sets the maximum size of the extracted text fragment, `250` by default
- `luc:snippetSpanOpen`: text to insert before the highlighted text, `<em>` by default
- `luc:snippetSpanClose`: text to insert after the highlighted text, `</em>` by default

The option predicates are set on the query instance, much like the `luc:query` predicate.

4.3.3.9 Total hits

You can get the total number of matching Lucene documents (hits) by using the `luc:totalHits` predicate. For the `my_index` connector instance used above, a query for all wines made in 2012 might look like this:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?totalHits {
    ?r a luc-index:my_index ;
       luc:query "year:2012" ;
       luc:totalHits ?totalHits .
}
```

As there are three wines made in 2012, the value `3` (of type `xdd:long`) binds to `?totalHits`.

As you see above, you can omit the return of any of the matching entities. This can be useful if there are many hits and you want to calculate pagination parameters.

4.3.4 List of creation parameters

The creation parameters define how the `luc:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object, where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the Create Connector user interface in the GraphDB Workbench without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal internal data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to make a Lucene index from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `luc:graph`, where the prefix `luc:`

represents the IRI `http://www.ontotext.com/connectors/lucene#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *readonly mode*.

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
INSERT {
    GRAPH luc:graph {
        ...
    }
} WHERE {
        ...
};

PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>
INSERT DATA {
    luc-index:my_index luc:createConnector '''
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponding to a single predicate with a field name the same as the predicate, so you must use escaping when issuing Lucene queries.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`analyzer` (string), optional, specifies Lucene analyzer The Lucene Connector supports custom Analyzer implementations. You can specify these using the `analyzer` parameter, whose value must be a fully qualified name of a class that extends `org.apache.lucene.analysis.Analyzer`. The class requires either a default constructor or a constructor with exactly one parameter of type `org.apache.lucene.util.Version`. For example, these two classes are valid implementations:

```
package com.ontotext.example;

import org.apache.lucene.analysis.Analyzer;

public class FancyAnalyzer extends Analyzer {
    public FancyAnalyzer() {
        ...
    }
    ...
}
```

```
package com.ontotext.example;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.util.Version;

public class SmartAnalyzer extends Analyzer {
    public SmartAnalyzer(Version luceneVersion) {
        ...
    }
    ...
}
```

You can then use `FancyAnalyzer` and `SmartAnalyzer` by specifying their fully qualified names. For example:

```
...
    "analyzer": "com.ontotext.example.SmartAnalyzer",
...
```

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type. See also the examples in *General full-text search with the connectors*.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual but you can map only some of the languages represented in the literal values. You can do this by specifying a list of language ranges to be matched to the language tags of literals according to RFC 4647, Section 3.3.1. Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to Lucene The fields define exactly what parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in Lucene. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

- **`fieldName` (string), required, the name of the field in Lucene** The name of the field defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field name is used at query time to refer to the field. There are few restrictions on the allowed characters in a field name, but to avoid unnecessary escaping (which depends on how Lucene parses its queries), we recommend keeping the field names simple.
- **`fieldNameTransform` (one of `none`, `predicate` or `predicate.localName`), optional, `none` by default** Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

  See *Indexing all literals in distinct fields* for an example.
- **`propertyChain` (list of IRI), required, defines the property chain to reach the value** The property chain (`propertyChain`) defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple and so on. In this model, a property chain with a single element corresponds to a direct

property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

See *Copy fields* for defining multiple fields with the same property chain.

See *Multiple property chains per field* for defining a field whose values are populated from more than one property chain.

See *Indexing language tags* for defining a field whose values are populated with the language tags of literals.

See *Indexing the IRI of an entity* for defining a field whose values are populated with the IRI of the indexed entity.

See *Wildcard literal indexing* for defining a field whose values are populated with literals regardless of their predicate.

- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.
- **`defaultValue` (string), optional, specifies a default value for the field** Provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or a IRI. It has no default value.
- **`indexed` (boolean), optional, default `true`** If indexed, a field is available for Lucene queries. `true` by default.

  This option corresponds to Lucene's field option `"indexed"`.
- **`stored` (boolean), optional, default `true`** Whether fields are stored in Lucene. `true` by default.

  This options corresponds to Lucene's property `"stored"`.
- **`analyzed` (boolean), optional, default `true`** When literal fields are indexed in Lucene, they will be analyzed according to the analyzer settings. Should you require that a given field is not analyzed, you may use `"analyzed"`. This option has no effect for IRIs, which are never analyzed. `true` by default.

  This option corresponds to Lucene's property "tokenized".
- **`multivalued` (boolean), optional, default `true`** RDF properties and synchronized fields may have more than one value. If `"multivalued"` is set to `true`, all values will be synchronized to Lucene. If set to `false`, only a single value will be synchronized. `true` by default.
- **`ignoreInvalidValues` (boolean), optional, default `false`** Per-field option that controls what happens when a value cannot be converted to the requested (or previously detected) type. `False` by default.

  Example use: when an invalid date literal like `"2021-02-29"^^xsd:date` (2021 is not a leap year) needs to be indexed as a date, or when an IRI needs to be indexed as a number.

  > **Note:** Some conversions are always valid: any literal to an FTS field, any non-literal (IRI, blank node, embedded triple) to a non-analyzed field. When `true`, such values will be skipped with a note in the logs. When `false`, such values will break the transaction.

- **`facet` (boolean), optional, default `true`** Whether Lucene's index can be used for faceted searches. This is controlled by the Boolean option "facet". `true` by default. Fields that are not synchronized for faceting are also not available for faceted search.
- **`datatype` (string), optional, the manual datatype override** By default, the Lucene GraphDB Connector uses datatype of literal values to determine how they must be mapped to Lucene types. For more information on the supported datatypes, see *Datatype mapping*.

  You can override the datatype mapping with the `"datatype"` parameter, which can be specified per field. The value of `"datatype"` can be any of the `xsd:` types supported by the automatic mapping.

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

4.3.4.1 Special field definitions

Copy fields

It is often convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases such as faceting or sorting vs full-text search. The Lucene GraphDB Connector has explicit support for fields that copy their value from another field. To do this, specify a single element in the property chain of the form `@otherFieldName`, where `otherFieldName` is another non-copy field. Take the following example:

```
...
  "fields": [
    {
      "fieldName": "grape",
      "facet": false,
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzed": true,
    },
    {
      "fieldName": "grapeFacet",
      "propertyChain": [
        "@grape"
      ],
      "analyzed": false,
    }
  ]
...
```

The snippet creates an analyzed field “grape” and a non-analyzed field “grapeFacet”; both fields are populated with the same values and “grapeFacet” is defined as a copy field that refers to the field “facet”.

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in Lucene) with more than one property chain — for example, the concept of “name” could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in Lucene you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.example.com/demo#canonicalName"
      ],
```

(continues on next page)

(continued from previous page)

```
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.example.com/demo#historicalName"
      ]
      ...
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in Lucene.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old` you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:createConnector '''
    {
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "lang()"
           ]
         }
      ],
    }
```

(continues on next page)

(continued from previous page)

```
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.example.com/demo#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:createConnector '''
    {
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "graph()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.example.com/demo#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:createConnector '''
    {
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
```

(continues on next page)

(continued from previous page)

```
          ]
        },
        {
          "fieldName": "feature",
          "propertyChain": [
            "http://www.example.com/demo#feature",
            "localName()"
          ]
        }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.example.com/demo#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
    luc-index:my_index luc:createConnector '''
{
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
```

(continues on next page)

(continued from previous page)

```
      "fieldName": "entityId",
      "propertyChain": [
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

4.3.5 Datatype mapping

The Lucene GraphDB Connector maps different types of RDF values to different types of Lucene values according to the basic type of the RDF value (IRI or literal) and the datatype of literals. The autodetection uses the following mapping:

RDF value	RDF datatype	Indexed in Lucene as
IRI	n/a	`Field` (not tokenized)
literal	any type not explicitly mentioned below	`Field` (tokenized, with term vectors)
literal	`xsd:boolean`	`Field` (not tokenized), with the values `true` and `false`
literal	`xsd:double`	`DoublePoint`
literal	`xsd:float`	`FloatPoint`
literal	`xsd:long`	`LongPoint`
literal	`xsd:int`	`LongPoint`
literal	`xsd:dateTime`	`Field` (not tokenized), padded string with second precision
literal	`xsd:date`	`Field` (not tokenized), padded string with day precision

The datatype mapping can be affected by the synchronization options. For example, a non-analyzed field that has `xsd:long` values is indexed as a non-tokenized `Field`.

Note: For any given field the automatic mapping uses the first value it sees. This works fine for clean datasets but might lead to problems if your dataset has non-normalized data — for example, the first value has no datatype but other values do.

Because of this, it's best to set the *datatype* to a fixed value such as `xsd:date`.

Please note that the commonly used `xsd:integer` and `xsd:decimal` datatypes are not indexed as numbers because they represent infinite precision numbers. You can override that by using the *datatype* option to cast them to `xsd:long`, `xsd:double`, or `xsd:float` as appropriate.

4.3.5.1 Date and time conversion

RDF and Lucene use different models to represent dates and times. Lucene stores values as offsets in seconds for sorting, or as padded ISO 8601 strings for range search. For example, `"2020-03-23T12:34:56"^^xsd:dateTime` will be stored as the string `20200323123456`.

Years in RDF values use the XSD format and are *era* years, where positive values denote the common era and negative values denote years before the common era. There is no year zero.

Years in padded string date and time Lucene values use the ISO format and are *proleptic* years: positive values denote years from the common era with any previous eras just going down by one mathematically so there is year zero.

In short:

- year 2020 CE = year 2020 in XSD = year 2020 in ISO.
- …
- year 1 CE = year 1 in XSD = year 1 in ISO.
- year 1 BCE = year -1 in XSD = year 0 in ISO.
- year 2 BCE = year -2 in XSD = year -1 in ISO.
- …

All years coming from RDF literals will be converted to ISO before indexing in Lucene.

Note: Range search will not work as expected with negative years. This is a limitation of storing the date and time values as strings.

XSD date and time values support timezones. In order to have a unified view over values with different timezones, all `xsd:dateTime` values will be normalized to the UTC time zone before indexing.

In addition to that, XSD defines the lack of a timezone as *undetermined*. Since we do not want to have any undetermined state in the indexing system, GraphDB defines the undetermined time zone as UTC. For example, `"2020-02-14T12:00:00"^^xsd:dateTime` is equivalent to `"2020-02-14T12:00:00Z"^^xsd:dateTime` (Z is the UTC time zone, also known as +00:00).

Also note that XSD dates may have a timezone, which leads to additional complications. For example, `"2020-01-01+02:00"^^xsd:date` (the date 1 January 2020 in the +02:00 timezone) will be normalized to `2019-12-31T22:00:00Z` (a different day!) if GraphDB followed strict timezone adherence. We have chosen to ignore the timezone on any values that do not have an associated time value. For example, both of the following values will be treated as if they specified UTC as their timezone:

- `"2020-02-15+02:00"^^xsd:date`
- `"2020-05-08-05:00"^^xsd:date`

4.3.6 Entity filtering

The Lucene connector supports three kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to Lucene if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable.

4.3.6.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

See also *Migrating from GraphDB 9.x*.

4.3.6.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on original RDF values before they are converted to Lucene values using *datatype mapping*.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`

Continued on next page

Table 4 – continued from previous page

Operator	Meaning
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible such as `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height` has a value of `"150"^^xsd:int` and `dateOfBirth` has a
	value of `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

Continued on next page

Table 4 – continued from previous page

Operator	Meaning

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.3.6.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.3.6.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.3.6.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.example.com/demo#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:createConnector '''
    {
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": ["http://www.example.com/demo#name"]
         },
         {
           "fieldName": "city",
           "propertyChain": ["http://www.example.com/demo#city"],
           "valueFilter": "$this = \\"London\\""
         }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
         {
           "fieldName": "city",
           "propertyChain": ["http://www.example.com/demo#city"],
           "defaultValue": "London"
         }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.3.6.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.example.com/demo2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
    rdf:type example2:Person ;
```

(continues on next page)

(continued from previous page)

```
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to Lucene, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  luc-index:my_index luc:createConnector '''
    {
      "types": ["http://www.example.com/demo2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
         {
           "fieldName": "taggedWithPerson",
           "propertyChain": ["http://www.example.com/demo2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo2#Person>"
         },
         {
           "fieldName": "taggedWithLocation",
```

(continues on next page)

(continued from previous page)

```
          "propertyChain": ["http://www.example.com/demo2#taggedWith"],
          "valueFilter": "$this -> type = <http://www.example.com/demo2#Location>"
        }
      ]
    }
  ''' .
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

You can check this by issuing a faceted search for `taggedWithLocation` and `taggedWithPerson`:

```
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?facetName ?facetValue ?facetCount {
  ?search a luc-index:my_index ;
      luc:facetFields "taggedWithLocation,taggedWithPerson" ;
      luc:facets [
        luc:facetName ?facetName ;
        luc:facetValue ?facetValue ;
        luc:facetCount ?facetCount
      ]
}
```

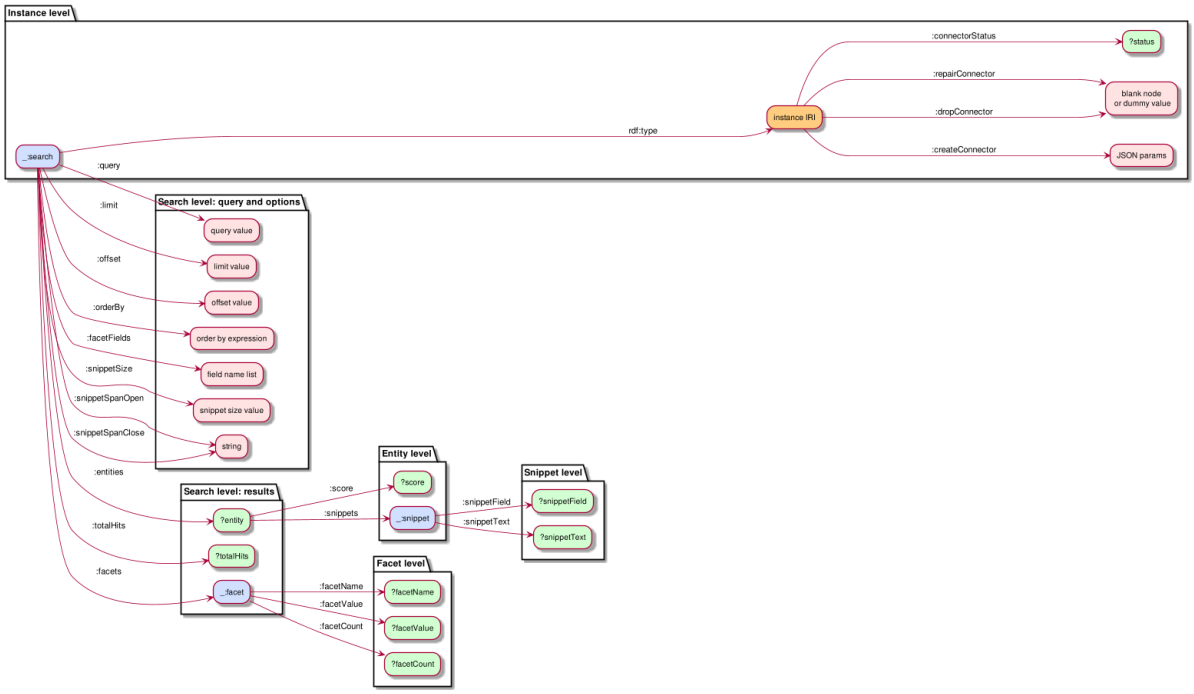
If the filter was applied, you should get only `:Berlin` for `taggedWithLocation` and only `:Einstein` and `:Mozart` for `taggedWithPerson`:

facetName	facetValue	facetCount
taggedWithLocation	`http://www.example.com/demo2#Berlin`	3
taggedWithPerson	`http://www.example.com/demo2#Mozart`	2
taggedWithPerson	`http://www.example.com/demo2#Einstein`	1

4.3.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to. For example, to retrieve entities, you need to use `:entities` on a search instance, and to retrieve snippets, you need to use `:snippets` on an entity.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.3.8 Caveats

4.3.8.1 Order of control

Even though SPARQL per se is not sensitive to the order of triple patterns, the Lucene GraphDB Connector expects to receive certain predicates before others so that queries can be executed properly. In particular, predicates that specify the query or query options must come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.

4.3.9 Upgrading from previous versions

4.3.9.1 Migrating from GraphDB 9.x

GraphDB 10.0 introduces major changes to the filtering mechanism of the connectors. Existing connector instances will not be usable and attempting to use them for queries or updates will throw an error.

If your GraphDB 9.x (or older) connector definitions **do not** include an entity filter, you can simply *repair them*.

If your GraphDB 9.x (or older) connector definitions **do** include an entity filter with the `entityFilter` option, you need to rewrite the filter with one of the current *filter types*:

1. Save your existing connector definition.
2. Drop the connector instance.

3. In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter. Rewrite the filters as follows.

   Rule of thumb:

   - If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

   So if we take the example:

   ```
   ?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
   ```

   It needs to be rewritten like this:

   - Per-field rule on field `location`: `$this = <urn:Europe>`
   - Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
   - Top-level document filter: `BOUND(?location)`
4. Recreate the connector instance using the new definition.

4.4 Solr GraphDB Connector

Note: This feature requires a *GraphDB Enterprise license*.

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Solr but with the additional benefit of staying automatically up to date with the GraphDB repository data.

Note: GraphDB supports *full-text search* options as well.

The Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

The main features of the GraphDB Connectors are:

- maintenance of an index that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entities synchronization defined by:
  - a list of fields (on the Solr side) and property chains (on the GraphDB side) whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization (the default is all languages)
  - additional filtering by property and value.
- full-text search using native Solr queries
- snippet extraction: highlighting of search terms in the search result
- faceted search
- sorting by any preconfigured field

- paging of results using `offset` and `limit`
- custom mapping of RDF types to Solr types

Each feature is described in detail below.

4.4.1 Usage

All interactions with the Solr GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the Solr GraphDB Connector, this is `http://www.ontotext.com/connectors/solr#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/solr#createConnector`) to create a connector instance for Solr.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For Solr, the instance IRI is `http://www.ontotext.com/connectors/solr/instance#`.

> **Warning:** Deleting the repository does not remove the indexes in Solr. In order to create a connector instance with the same name as one of the instances in the deleted repository, you may need to delete the corresponding Solr index first.

> **Warning:** Changing the Solr URL will not reindex the data automatically. This might lead to issues such as the inability to create another connector with the same name as the deleted one. If you need the data to be reindexed, you can *repair* the connector instance.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito and Rozova as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .
```

(continues on next page)

(continued from previous page)

```
wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

4.4.2 Setup and maintenance

4.4.2.1 Prerequisites

Tip: In versions prior to 9.1 of SolrJ, Solr claimed that the client is backward compatible. However, the documentation for Solr 9.1 does not mention anything about version compatibility. Our tests indicate that the SolrJ 9.1 can connect to Solr server 8.x and newer.

Solr core creation To create new Solr cores on the fly, you have to use the custom admin handler provided with the Solr Connector.

1. Copy the `solr-core-admin-handler.jar` file from the `/tools` to the `/configs/solr-home/` directory of the GraphDB distribution.
2. To start Solr, execute:

```
<path-to-solr-distribution>/bin/solr start -p 8934 -s /<path-to-solr-home>
```

Solr schema setup To use the connector, the core's schema from which the configuration will be copied (most of the time named collection1) must be configured to allow schema modifications. See "Managed Schema Definition in SolrConfig" on page 409 of the Apache Solr Reference Guide.

A good starting point is the configuration from `example-schemaless` in the Solr distribution.

Third-party component versions The Solr GraphDB Connector in GraphDB 10.4.0 uses Solr version 8.11.2.

The Solr GraphDB Connector in GraphDB 10.4.1 and newer uses Solr version 9.4.0.

4.4.2.2 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- a Solr instance to synchronize to
- classes to synchronize
- properties to synchronize

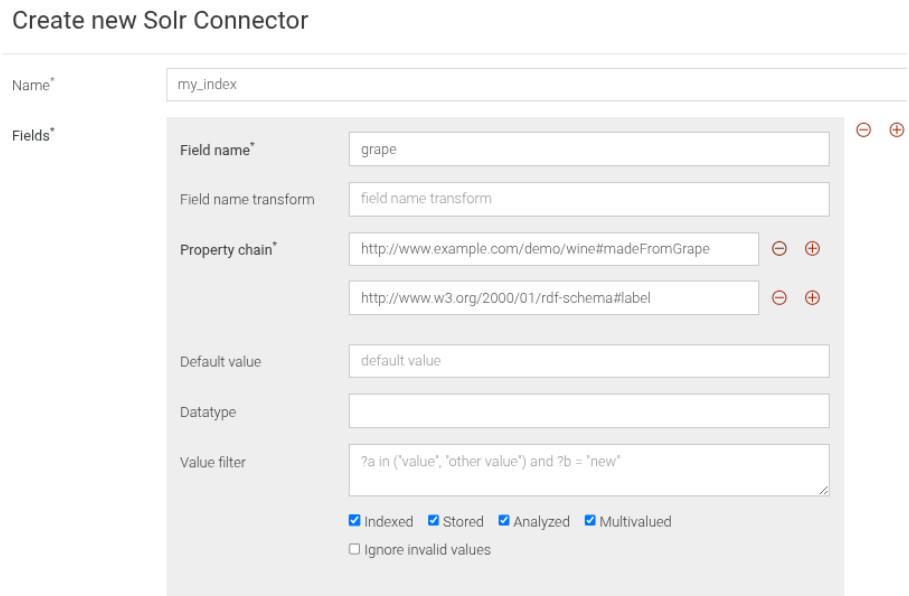
The configuration data must be provided as a JSON string representation and passed together with the create command.

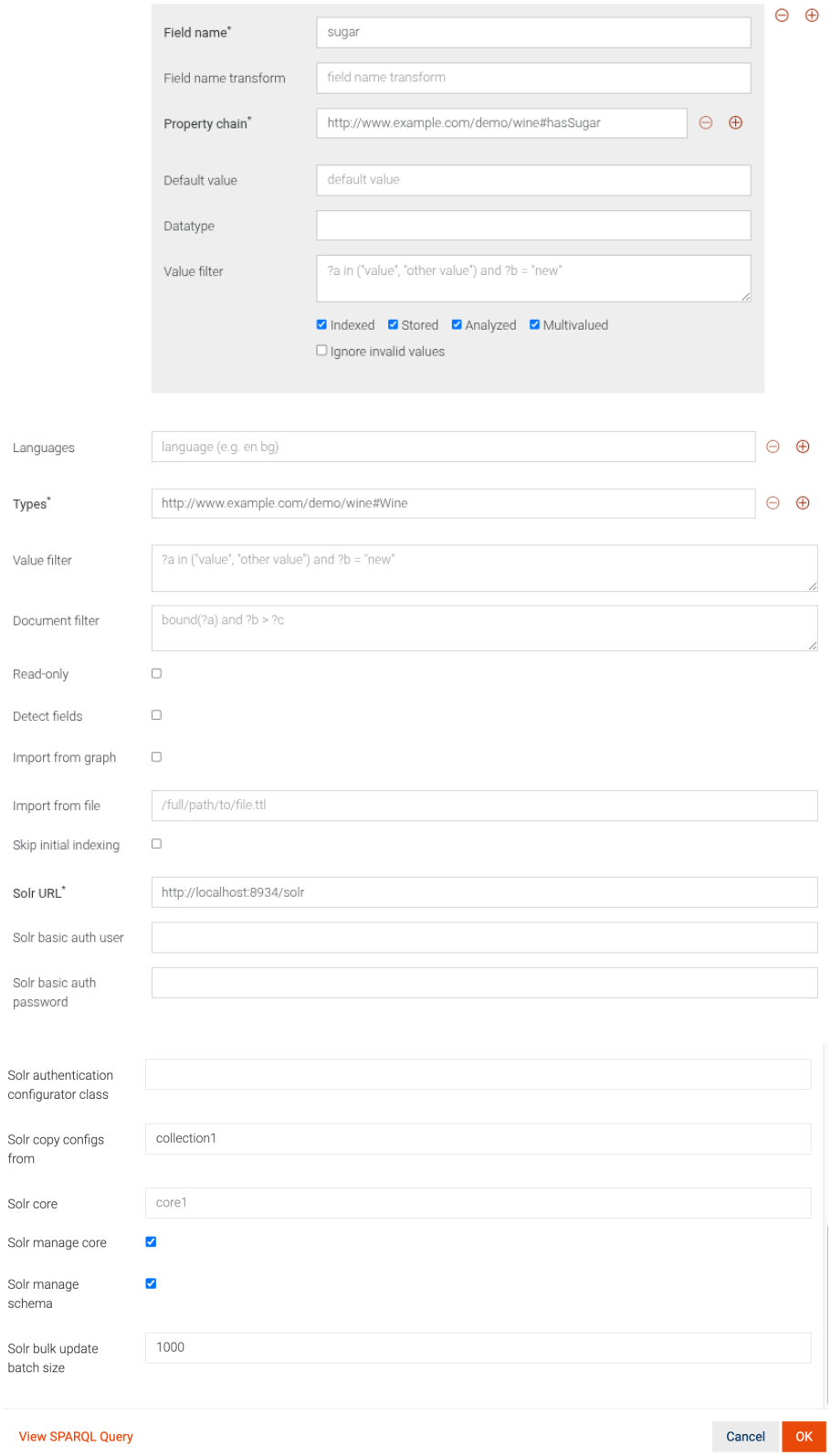
You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill in the configuration form.





1. Execute the `CREATE` statement from the form by clicking *OK*. Alternatively, you can view its SPARQL query by clicking *View SPARQL Query*, and then copy it to execute it manually or integrate it in automation scripts.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `solr:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
    solr-index:my_index solr:createConnector '''
{
  "solrUrl": "http://localhost:8983/solr",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false,
      "multivalued": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ]
}
''' .
}
```

Note: One of the fields has `"multivalued": false`. This is explained further under *Sorting*.

The above command creates a new Solr connector instance that connects to the Solr instance accessible at port `8983` on the localhost as specified by the `"solrUrl"` key.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to Solr. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: "grape", "sugar", and "year". The field names are used later in the queries.

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine's `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to

the wine.

The fields `sugar` and `year` contain discrete values such as *medium*, *dry*, *2012*, and *2013*, so it is best to specify the option `analyzed: false` as well. See `analyzed` in *Defining fields* for more information.

Schema and core management

By default, GraphDB manages (create, delete or update if needed) the Solr core and the Solr schema. This makes it easier to use Solr as everything is done automatically. This behavior can be changed by the following options:

- `manageCore`: if `true`, GraphDB manages the core. `true` by default.
- `manageSchema`: if `true`, GraphDB manages the schema. `true` by default.

The automatic core management requires the custom Solr admin handler provided with the GraphDB distribution. For more information, see *Solr core creation*.

Note: If either of the options is set to `false`, you have to create, update or remove the `core/schema` manually and, in case Solr is misconfigured, the connector instance will not function correctly.

Using a non-managed schema

The present version provides no support for changing some advanced options, such as stop words, on a per-field basis. The recommended way to do this for now is to manage the schema yourself and tell the connector to just sync the object values in the appropriate fields. Here is an example:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
    solr-index:my_index solr:createConnector '''
{
  "solrUrl": "http://localhost:8983/solr",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false,
      "multivalued": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
```

(continues on next page)

(continued from previous page)

```
  ],
  "manageSchema": "false"
}
''' .
}
```

This creates the same connector instance as above but it expects fields with the specified field names to be already present in the core as well as some internal GraphDB fields. For the example, you must have the following fields:

Field name	Solr config
`_graphdb_id`	`<field name="_graphdb_id" type="string" indexed="true" stored="true" required="true" multiValued="false"/>`
`grape`	`<field name="grape" type="text_general" indexed="true" stored="true" multiValued="true"/>`
`sugar`	`<field name="sugar" type="text_general" indexed="true" stored="true" multiValued="false"/>`
`year`	`<field name="year" type="tints" indexed="true" stored="true" multiValued="true"/>`

`_graphdb_id` is used internally by GraphDB and is always required.

Working with secured Solr

GraphDB allows the access of a secured Solr instance by passing the arbitrary parameters.

To setup basic user authentication configuration in GraphDB Solr Connector, you need to configure the `solrBasicAuthUser` and `solrBasicAuthPassword` parameters.

```
...
      solr-index:my_index conn:createConnector '''
    {
        "hasProperty": "http://www.w3.org/2000/01/rdf-schema#comment",
        "solrUrl": "${validSolrUrl}",
        "solrUrl": "http://localhost:9090/solr",
        "solrBasicAuthUser": "solr",
        "solrBasicAuthPassword": "SolrRocks",
        "fields": [
...
```

When you create a new Solr Connector in GraphDB Workbench, you need to add values for the `solrBasicAuthUser` and `solrBasicAuthPassword` options.

Instead of supplying the username and password as part of the connector instance configuration, you can also implement a custom authenticator class and set it via the `authenticationConfiguratorClass` option. See these connector authenticator examples for more information and example projects that implement such a custom class.

For more information about securing Solr, see the documentation for Solr: Enable Basic Authentication.

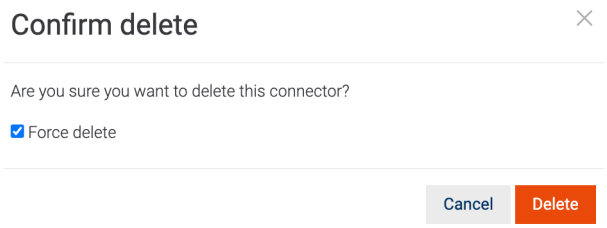
4.4.2.3 Dropping a connector instance

Dropping a connector instance removes all references to its external store from GraphDB as well as the Solr core associated with it.

The drop command is triggered by a SPARQL `INSERT` with the `solr:dropConnector` predicate where the name of the connector instance must be in the subject position. For example, the following removes the connector `my_index`:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index solr:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon 🗑, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.4.2.4 Retrieving the create options for a connector instance

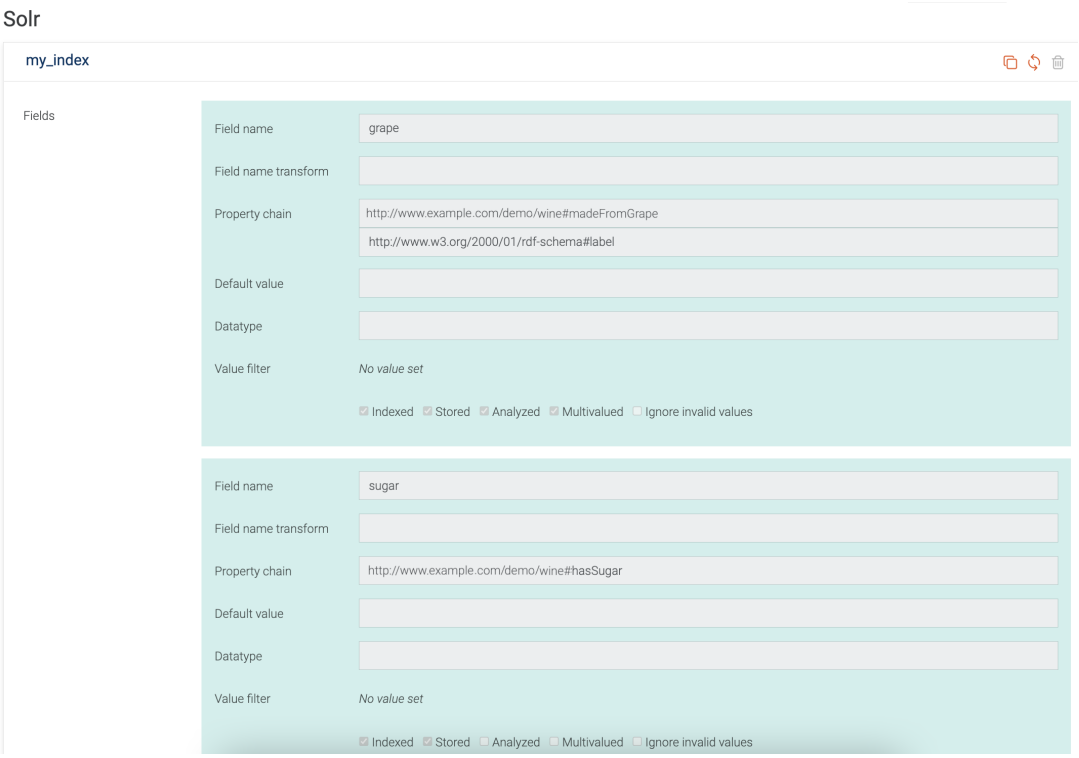
You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?createString {
  solr-index:my_index solr:listOptionValues ?createString .
}
```

4.4.2.5 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* 🔄 or *Delete* 🗑 icons to perform these operations. Click the *Copy* 📋 icon to copy the connector definition query to your clipboard.



With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>

SELECT ?cntUri ?cntStr {
  ?cntUri solr:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/solr/instance#my_index>`, while `?cntStr` is bound to a string representing the local name such as “my_index” in this example.

4.4.2.6 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>

SELECT ?cntUri ?cntStatus {
  ?cntUri solr:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.4.3 Working with data

4.4.3.1 Adding, updating, and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` update requests. This is achieved by intercepting all changes in the plugin and determining which Solr documents need to be updated.

4.4.3.2 Simple queries

Once a connector instance has been created, you can use SPARQL to query data from it. For each matching Solr document, the connector instance returns the document subject. In its simplest form, you query that data using a `SELECT` query that includes a Solr query as the object of the `solr:query` predicate:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity {
  ?search a solr-index:my_index ;
      solr:query "grape:cabernet" ;
      solr:entities ?entity .
}
```

The result of this query binds `?entity` to the two wines made from grapes that have "cabernet" in their name: `:Yoyowine` and `:Franvino`.

Note: You must use the field names you chose when you created the connector instance. They can be identical to the property IRIs but you must escape any special characters according to what Solr expects.

The creation of such a query follows these basic steps:

1. Get a query instance of the requested connector instance by using the RDF notation `"X a Y" (= X rdf:type Y)`, where `X` is a variable and `Y` is a connector instance IRI. `X` is bound to a query instance of the connector instance.
2. Assign a query to the query instance by using the system predicate `solr:query`.
3. Request the matching entities through the `solr:entities` predicate.

You can also provide per-query search options by using one or more option predicates. The option predicates are described in detail below.

4.4.3.3 Raw queries

To access a Solr query parameter that is not exposed through a special predicate, use a raw query. Instead of providing a full-text query in the `:query` part, specify raw Solr parameters. For example, to sort the facets in a different order than described in facet.sort, execute the following query:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity {
  ?search a solr-index:my_index ;
      solr:query '''
      {
          "facet":"true",
          "indent":"true",
          "facet.sort":"index",
```

(continues on next page)

(continued from previous page)

```
            "q":"*:*",
            "wt":"json"
        }
        ''' ;
        solr:entities ?entity .
}
```

You can get these parameters when you do your query from the admin interface in Solr, or from the response payload (where they are included). The query parameters from the select endpoint are also supported in Solr. Here is an example:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity {
  ?search a solr-index:my_index ;
      solr:query '''q=*%3A*&wt=json&indent=true&facet=true&facet.sort=index''' ;
      solr:entities ?entity .
}
```

Note: You have to specify `q=` as the first parameter as it is used for detecting the raw query.

Combining Solr results with GraphDB data

You can use the bound `?entity` variable in other SPARQL triple patterns to build complex queries that join to or fetch additional data from GraphDB. For example, to see the actual grapes in the matching wines as well as the year they were made:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?entity ?grape ?year {
  ?search a solr-index:my_index ;
      solr:query "grape:cabernet" ;
      solr:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
  ?entity wine:hasYear ?year
}
```

The result looks like this:

	entity	grape	year
1	wine:Yoyowine	wine:CabernetSauvignon	"2013"^^xsd:integer
2	wine:Franvino	wine:Merlot	"2012"^^xsd:integer
3	wine:Franvino	wine:CabernetFranc	"2012"^^xsd:integer

Note: `:Franvino` is returned twice because it is made from two different grapes, both of which are returned.

Entity match score

It is possible to access the match score returned by Solr with the `score` predicate. As each entity has its own score, the predicate should come at the entity level. For example:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity ?score {
  ?search a solr-index:my_index ;
      solr:query "grape:cabernet" ;
      solr:entities ?entity .
  ?entity solr:score ?score
}
```

The result looks like this but the actual score might be different as it depends on the specific Solr version:

	entity	score
1	wine:Yoyowine	"0.3979403078556061"^^xsd:double
2	wine:Franvino	"0.33036553859710693"^^xsd:double

4.4.3.4 Basic facet queries

You can also query facets with your connector instance. For example, with the sample `my_index` connector described above:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?facetName ?facetValue ?facetCount WHERE {
  # note empty query is allowed and will just match all documents, hence no :query
  ?r a solr-index:my_index ;
    solr:facetFields "year,sugar" ;
    solr:facets [
     solr:facetName ?facetName;
     solr:facetValue ?facetValue;
     solr:facetCount ?facetCount
   ]
}
```

It is important to specify the facet fields by using the `facetFields` predicate. Its value is a simple comma-delimited list of field names. In order to get the faceted results, use the `solr:facets` predicate. Because each facet has three components (name, value and count), the `solr:facets` predicate returns multiple nodes that can be used to access the individual values for each component through the predicates `facetName`, `facetValue`, and `facetCount`.

The resulting bindings look like the following:

	facetName	facetValue	facetCount
1	"year"	"2012"	"3"^^xsd:long
2	"year"	"2013"	"2"^^xsd:long
3	"sugar"	"dry"	"3"^^xsd:long
4	"sugar"	"medium"	"2"^^xsd:long

You can easily see that there are three wines produced in 2012 and two in 2013. You also see that three of the wines are dry, while two are medium. However, it is not necessarily true that the three wines produced in 2012 are the same as the three dry wines as each facet is computed independently.

Tip: Faceting by analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens, and faceting uses each token to create a faceting bucket. For example, "North America" and "Europe" produce three buckets: "north", "america" and "europe", corresponding to each token in the two values. If you need to facet by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.4.3.5 Advanced facet and aggregation queries

While basic faceting allows for simple counting of documents based on the discrete values of a particular field, there are more complex faceted or aggregation searches in Solr. The Solr GraphDB Connector provides a mapping from Solr results to RDF results but no mechanism for specifying the queries other than executing a *Raw queries*.

Supported Solr facets and aggregations

The Solr GraphDB Connector supports mapping of range, interval, and pivot facets.

Tip: For more information, refer to the documentation of Solr.

RDF mapping of the results

The results are accessed through the predicate `aggregations` (much like the basic facets are accessed through `facets`). The predicate binds multiple blank nodes that each contain a single aggregation bucket. The individual bucket items can be accessed through these predicates:

predicate	meaning	Solr counterpart
`:name`	Bucket name	`getName()`
`:key`	Key or value associated with the bucket	`getValue()` or `getKey()`
`:count`	Count of documents in the bucket	`getCount()`
`:from`	Start of range (RangeFacet)	`getStart()`
`:to`	End of range (RangeFacet)	`getEnd()`
`:rangeGap`	Gap of range (RangeFacet)	`getGap()`
`:beforeCount`	Count of documents before the first range (RangeFacet)	`getBefore()`
`:afterCount`	Count of documents after the first range (RangeFacet)	`getAfter()`
`:betweenCount`	Count of documents within all ranges (RangeFacet)	`getBetween()`
`:parent`	Pivot facets: points to the parent (upper level) blank node	
`:level`	Pivot facets: level number where 1 is the uppermost level and the following levels are 2, 3 and so on	
`:levelName`	Pivot facets: level name	`getField()`

4.4.3.6 Sorting

You can sort the entities returned by a connector query according to one or more fields. To do this, assign a comma-delimited list of sort fields to the `orderBy` predicate. Each field can be prefixed with a minus to indicate sorting in descending order. For example:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>
SELECT ?entity ?sugar{
  ?search a solr-index:my_index ;
      solr:query "year:2013" ;
      solr:orderBy "-sugar" ;
      solr:entities ?entity.
  ?entity wine:hasSugar ?sugar
}
```

The result of this sample query contains wines produced in 2013 sorted according to their sugar content in descending order:

	entity	sugar
1	wine:Rozova	"medium"
2	wine:Yoyowine	"dry"

By default, entities are sorted according to their matching score in descending order.

Note: If you join the entity from the connector query to other triples stored in GraphDB, GraphDB might scramble the order. Avoid this by using SPARQL's `ORDER BY` instruction.

Tip: Sorting by an analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens and sorting uses the least (in the lexicographical sense) token. For example, "North America" will be sorted before "Europe" because the token "america" is lexicographically smaller than the token "europe". If you need to sort by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

Note: Solr imposes an additional requirement on fields used for sorting. They must be defined with `multivalued = false`.

4.4.3.7 Limit and offset

Limit and offset are supported on the Solr side of the query. You can set these values using the predicates `limit` and `offset`. Consider this example in which an offset of `1` and a limit of `1` are specified:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity {
  ?search a solr-index:my_index ;
      solr:query "sugar:dry" ;
      solr:offset "1" ;
      solr:limit "1" ;
      solr:entities ?entity .
}
```

`offset` is counted from 0. The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:

	entity
1	wine:Yoyowine
2	wine:Franvino
3	wine:Blanquito

Note: The specific order in which GraphDB returns the results depends on how Solr returns the matches, unless sorting is specified.

4.4.3.8 Snippet extraction

Snippet extraction lets you retrieve highlighted snippets of text that match a query. You can access the snippets by using the the dedicated predicate `solr:snippets`. This binds a blank node that in turn provides the actual snippets via the predicates `solr:snippetField` and `solr:snippetText`. The predicate snippets must be attached to the entity, as each entity has a different set of snippets. For example, in a search for Cabernet:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a solr-index:my_index ;
      solr:query "grape:cabernet" ;
      solr:entities ?entity .
  ?entity solr:snippets ?snippet .
  ?snippet solr:snippetField ?snippetField ;
     solr:snippetText ?snippetText .
}
```

the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

	entity	snippetField	snippetText
1	wine:Yoyowine	"grape"	"Cabernet Sauvignon"
2	wine:Franvino	"grape"	"MerlotCabernet Franc"

Note: The actual snippets might be different depending on the specific Solr implementation.

You can tweak how the snippets are collected and composed by using the following option predicates:

- `solr:snippetSize`: sets the maximum size of the extracted text fragment, `250` by default
- `solr:snippetSpanOpen`: text to insert before the highlighted text, `<em>` by default
- `solr:snippetSpanClose`: text to insert after the highlighted text, `</em>` by default

The option predicates are set on the query instance, much like the `solr:query` predicate.

4.4.3.9 Total hits

You can get the total number of matching Solr documents (hits) by using the `solr:totalHits` predicate. For the `my_index` connector instance used above, a query for all wines made in 2012 might look like this:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?totalHits {
    ?r a solr-index:my_index ;
       solr:query "year:2012" ;
       solr:totalHits ?totalHits .
}
```

As there are three wines made in 2012, the value `3` (of type `xdd:long`) binds to `?totalHits`.

As you see above, you can omit the return of any of the matching entities. This can be useful if there are many hits and you want to calculate pagination parameters.

4.4.4 List of creation parameters

The creation parameters define how the `solr:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object, where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the Create Connector user interface in the GraphDB Workbench without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to make a Solr index from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `solr:graph`, where the prefix `solr:` represents the IRI `http://www.ontotext.com/connectors/solr#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *readonly mode*.

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
INSERT {
    GRAPH solr:graph {
        ...
    }
} WHERE {
        ...
};
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>
INSERT DATA {
    solr-index:my_index solr:createConnector '''
```

(continues on next page)

(continued from previous page)

```
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponding to a single predicate with a field name the same as the predicate, so you must use escaping when issuing Solr queries.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`solrUrl` (URL), required, Solr instance to sync to As Solr is a third-party service, you have to specify the URL on which it is running. The URL takes the form `http://hostname.domain:port/`. There is no default value. Can be updated at runtime with no need to rebuild the index.

`solrBasicAuthUser` (string), optional, the settings for supplying the authentication user No default value. Can be updated at runtime with no need to rebuild the index.

`solrBasicAuthPassword` (string), optional, the settings for supplying the authentication password A password is a string with a single value that is not logged or printed. No default value. Can be updated at runtime with no need to rebuild the index.

`bulkUpdateBatchSize` (integer), controls the maximum number of documents sent per bulk request. Default value is 1,000. Can be updated at runtime with no need to rebuild the index.

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type. See also the examples in *General full-text search with the connectors*.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual but you can map only some of the languages represented in the literal values. You can do this by specifying a list of language ranges to be matched to the language tags of literals according to RFC 4647, Section 3.3.1. Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to Solr The fields define exactly what parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in Solr. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

- **`fieldName` (string), required, the name of the field in Solr** The name of the field defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field name is used at query time to refer to the field. There are few restrictions on the allowed characters in a field name, but to avoid unnecessary escaping (which depends on how Solr parses its queries), we recommend keeping the field names simple.

- **`fieldNameTransform` (one of `none`, `predicate` or `predicate.localName`), optional, `none` by default** Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

  See *Indexing all literals in distinct fields* for an example.
- **`propertyChain` (list of IRIs), required, defines the property chain to reach the value** The property chain (`propertyChain`) defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple and so on. In this model, a property chain with a single element corresponds to a direct property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

  The IRI of the document will be synchronized to the special field `"id"` in Solr. You may use it to query Solr directly and retrieve the matching entity IRI.

  See *Copy fields* for defining multiple fields with the same property chain.

  See *Multiple property chains per field* for defining a field whose values are populated from more than one property chain.

  See *Indexing language tags* for defining a field whose values are populated with the language tags of literals.

  See *Indexing the IRI of an entity* for defining a field whose values are populated with the IRI of the indexed entity.

  See *Wildcard literal indexing* for defining a field whose values are populated with literals regardless of their predicate.
- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.
- **`defaultValue` (string), optional, specifies a default value for the field** Provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or a IRI. It has no default value.
- **`indexed` (boolean), optional, default `true`** If indexed, a field is available for Solr queries. `true` by default.

  This options corresponds to the property `"indexed"` in the Solr schema.
- **`stored` (boolean), optional, default `true`** Whether fields are stored in Solr. `true` by default.

  This option corresponds to the property `"stored"` in the Solr schema.
- **`analyzed` (boolean), optional, default `true`** When literal fields are indexed in Solr, they will be analyzed according to the analyzer settings. Should you require that a given field is not analyzed, you may use `"analyzed"`. This option has no effect for IRIs, which are never analyzed. `true` by default.

  This option affects the Solr type that is used for the field. True uses a type suitable for the values (i.e., text or numeric), while `false` uses the type `"string"`, which is never analyzed by Solr.
- **`multivalued` (boolean), optional, default `true`** RDF properties and synchronized fields may have more than one value. If `"multivalued"` is set to `true`, all values will be synchronized to Solr. If set to `false`, only a single value will be synchronized. `true` by default.

This option corresponds to the `"multiValued"` property in the Solr schema. Note that Solr cannot order results by multivalued fields so you need to adjust your options accordingly.

- **`ignoreInvalidValues` (boolean), optional, default `false`** Per-field option that controls what happens when a value cannot be converted to the requested (or previously detected) type. `False` by default.

  Example use: when an invalid date literal like `"2021-02-29"^^xsd:date` (2021 is not a leap year) needs to be indexed as a date, or when an IRI needs to be indexed as a number.

  Note: Some conversions are always valid: any literal to an FTS field, any non-literal (IRI, blank node, embedded triple) to a non-analyzed field. When `true`, such values will be skipped with a note in the logs. When `false`, such values will break the transaction.

- **`datatype` (string), optional, the manual datatype override** By default, the Solr GraphDB Connector uses datatype of literal values to determine how they must be mapped to Solr types. For more information on the supported datatypes, see *Datatype mapping*.

  You can override the datatype mapping with the "datatype" property, which can be specified per field. The value of "datatype" can be any of the xsd: types supported by the automatic mapping or a native Solr type prefixed by `native:`, e.g., both `xsd:long` and `native:tlongs` map to the `tlongs` type in Solr.

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

4.4.4.1 Updating parameters at runtime

As mentioned above, you can update the following connection parameters at runtime with no need to rebuild the index:

- `solrUrl`
- `bulkUpdateBatchSize`
- `solrBasicAuthUser`
- `solrBasicAuthPassword`

The following SPARQL update request demonstrates this by updating the user and password values:

```
PREFIX conn:<http://www.ontotext.com/connectors/solr#>
PREFIX inst:<http://www.ontotext.com/connectors/solr/instance#>
INSERT DATA {
inst:properIndex conn:updateConnector '''
  {
      "solrBasicAuthUser": "foo",
      "solrBasicAuthPassword": "bar"
  }
'''.
}
```

4.4.4.2 Special field definitions

Copy fields

It is often convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases such as faceting or sorting vs full-text search. The Solr GraphDB Connector has explicit support for fields that copy their value from another field. To do this, specify a single element in the property chain of the form `@otherFieldName`, where `otherFieldName` is another non-copy field. Take the following example:

```
...
  "fields": [
    {
      "fieldName": "grape",
      "facet": false,
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzed": true,
    },
    {
      "fieldName": "grapeFacet",
      "propertyChain": [
        "@grape"
      ],
      "analyzed": false,
    }
  ]
...
```

The snippet creates an analyzed field "grape" and a non-analyzed field "grapeFacet"; both fields are populated with the same values and "grapeFacet" is defined as a copy field that refers to the field "facet".

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in Solr) with more than one property chain — for example, the concept of "name" could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in Solr you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.ontotext.com/example#canonicalName"
      ],
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.ontotext.com/example#historicalName"
      ]
```

(continues on next page)

(continued from previous page)

```
      ...
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in Solr.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old` you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example,

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index :createConnector '''
    {
      "solrUrl": "http://localhost:8984/solr",
      "types": ["http://www.ontotext.com/example#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.ontotext.com/example#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.ontotext.com/example#name",
             "lang()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.ontotext.com/example#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index :createConnector '''
    {
      "solrUrl": "http://localhost:8983/solr",
      "types": ["http://www.ontotext.com/example#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.ontotext.com/example#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.ontotext.com/example#name",
             "graph()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.ontotext.com/example#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index solr:createConnector '''
    {
      "solrUrl": "http://localhost:8983/solr",
      "types": ["http://www.ontotext.com/example#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.ontotext.com/example#name"
           ]
         },
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "feature",
          "propertyChain": [
            "http://www.ontotext.com/example#feature",
            "localName()"
          ]
        }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.ontotext.com/example#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example,

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
    solr-index:my_index solr:createConnector '''
{
  "solrUrl": "http://localhost:8983/solr",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "entityId",
```

(continues on next page)

(continued from previous page)

```
      "propertyChain": [
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

Note: Note that GraphDB will also use the IRI of each entity as the ID of each document in Solr, which is represented by the field `id`.

4.4.5 Datatype mapping

The Solr GraphDB Connector maps different types of RDF values to different types of Solr values according to the basic type of the RDF value (IRI or literal) and the datatype of literals. The autodetection uses the following mapping:

RDF value	RDF datatype	Solr type
IRI	n/a	`string`
literal	any type not explicitly mentioned below	`text_general`
literal	with one of the language tags **en**, **de**, **es**, **ru**	`text_xx` where `xx` is language dependent
literal	`xsd:boolean`	`boolean`
literal	`xsd:double`	`pdouble` (single value), `pdoubles` (multivalued)
literal	`xsd:float`	`pfloat` (single value), `pfloats` (multivalued)
literal	`xsd:long`	`plong` (single value), `plongs` (multivalued)
literal	`xsd:int`	`pint` (single value), `pints` (multivalued)
literal	`xsd:dateTime`	`pdate` (single value), `pdates` (multivalued)
literal	`xsd:date`	`pdate` (single value), `pdates` (multivalued)
literal	`xsd:gYear`	`pdate` (single value), `pdates` (multivalued)
literal	`xsd:gYearMonth`	`pdate` (single value), `pdates` (multivalued)

The datatype mapping can be affected by the synchronization options. For example, a non-analyzed field that has `xsd:long` values does not use `plong` or `plongs` but `string` instead.

Note: For any given field the automatic mapping uses the first value it sees. This works fine for clean datasets but might lead to problems if your dataset has non-normalized data — for example, the first value has no datatype but other values do.

Because of this, it's best to set the *datatype* to a fixed value such as `xsd:date`.

Please note that the commonly used `xsd:integer` and `xsd:decimal` datatypes are not indexed as numbers because they represent infinite precision numbers. You can override that by using the *datatype* option to cast them to `xsd:long`, `xsd:double`, or `xsd:float` as appropriate.

4.4.5.1 Date and time conversion

RDF and Solr use slightly different models to represent dates and times, even though the values might look very similar.

Years in RDF values use the XSD format and are *era* years, where positive values denote the common era and negative values denote years before the common era. There is no year zero.

Years in Solr use the ISO 8601 format and are *proleptic* years: positive values denote years from the common era with any previous eras just going down by one mathematically so there is year zero.

In short:

- year 2020 CE = year 2020 in XSD = year 2020 in ISO.
- …
- year 1 CE = year 1 in XSD = year 1 in ISO.
- year 1 BCE = year -1 in XSD = year 0 in ISO.
- year 2 BCE = year -2 in XSD = year -1 in ISO.
- …

All years coming from RDF literals will be converted to ISO before indexing in Solr.

Both XSD and ISO date and time values support timezones. Solr requires all date and time values to be normalized to the UTC timezone, so the Solr connector will convert the values accordingly before sending them to Solr for indexing.

In addition to that, XSD defines the lack of a timezone as *undetermined*. Since we do not want to have any undetermined state in the indexing system, GraphDB defines the undetermined time zone as UTC. For example, `"2020-02-14T12:00:00"^^xsd:dateTime` is equivalent to `"2020-02-14T12:00:00Z"^^xsd:dateTime` (Z is the UTC time zone, also known as +00:00).

Also note that XSD dates and partial dates, e.g., `xsd:gYear` values, may have a timezone, which leads to additional complications. For example, `"2020+02:00"^^xsd:gYear` (the year 2020 in the +02:00 timezone) will be normalized to `2019-12-31T22:00:00Z` (the previous year!) if GraphDB followed strict timezone adherence. We have chosen to ignore the timezone on any values that do not have an associated time value. For example, all of the following values will be treated as if they specified UTC as their timezone:

- `"2020-02-15+02:00"^^xsd:date`
- `"2020-02+02:00"^^xsd:gYearMonth`
- `"2020+02:00"^^xsd:gYear`

4.4.6 Entity filtering

The Solr connector supports three kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to Solr if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable.

4.4.6.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

See also *Migrating from GraphDB 9.x*.

4.4.6.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on original RDF values before they are converted to Solr values using *datatype mapping*.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`

Continued on next page

Table 7 – continued from previous page

Operator	Meaning
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible such as `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height` has a value of `"150"^^xsd:int` and `dateOfBirth` has a
	value of `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

Continued on next page

Table 7 – continued from previous page

Operator	Meaning

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.4.6.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.4.6.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.4.6.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.ontotext.com/example#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index solr:createConnector '''
    {
      "solrUrl": "http://localhost:8983/solr",
      "types": ["http://www.ontotext.com/example#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": ["http://www.ontotext.com/example#name"]
         },
         {
           "fieldName": "city",
           "propertyChain": ["http://www.ontotext.com/example#city"],
           "valueFilter": "$this = \\"London\\""
         }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
         {
           "fieldName": "city",
           "propertyChain": ["http://www.ontotext.com/example#city"],
           "defaultValue": "London"
         }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.4.6.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.ontotext.com/example2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
```

(continues on next page)

(continued from previous page)

```
    rdf:type example2:Person ;
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to Solr, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  solr-index:my_index solr:createConnector '''
    {
      "solrUrl": "http://localhost:8983/solr",
      "types": ["http://www.ontotext.com/example2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
         {
           "fieldName": "taggedWithPerson",
           "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.ontotext.com/example2#Person>"
         },
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "taggedWithLocation",
          "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
          "valueFilter": "$this -> type = <http://www.ontotext.com/example2#Location>"
        }
      ]
    }
  ''' .
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

You can check this by issuing a faceted search for `taggedWithLocation` and `taggedWithPerson`:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?facetName ?facetValue ?facetCount {
  ?search a solr-index:my_index ;
      solr:facetFields "taggedWithLocation,taggedWithPerson" ;
      solr:facets [
        solr:facetName ?facetName ;
        solr:facetValue ?facetValue ;
        solr:facetCount ?facetCount
      ]
}
```

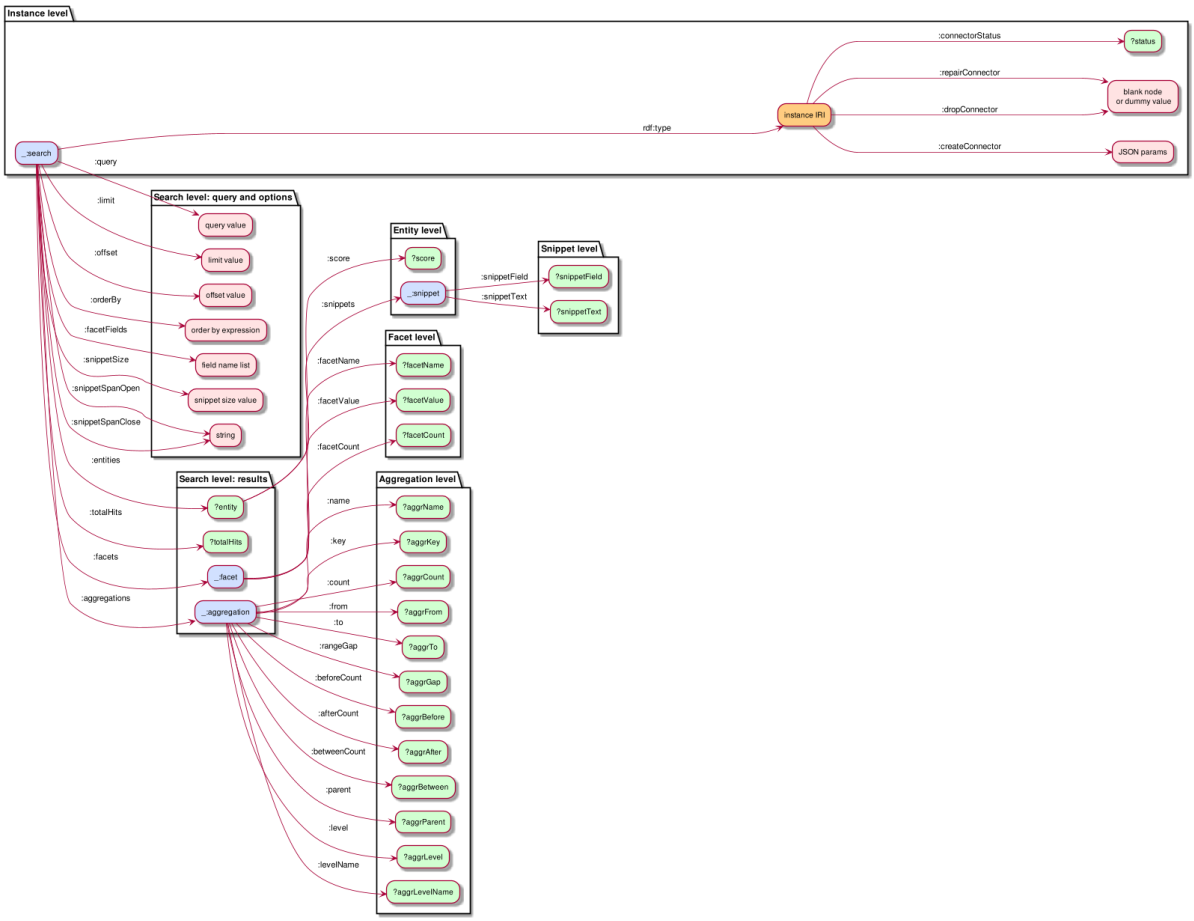
If the filter was applied, you should get only `:Berlin` for `taggedWithLocation` and only `:Einstein` and `:Mozart` for `taggedWithPerson`:

facetName	facetValue	facetCount
taggedWithLocation	`http://www.ontotext.com/example2#Berlin`	3
taggedWithPerson	`http://www.ontotext.com/example2#Mozart`	2
taggedWithPerson	`http://www.ontotext.com/example2#Einstein`	1

4.4.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to. For example, to retrieve entities, you need to use `:entities` on a search instance, and to retrieve snippets, you need to use `:snippets` on an entity.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.4.8 SolrCloud support

The Solr connector supports SolrCloud since the release of GraphDB 8.0/Connectors 6.0. SolrCloud is the distributed version of Solr, which offers features such as index sharding, better scaling, and fault tolerance. It uses Apache Zookeeper for distributed synchronization and central configuration of the Solr nodes. The Solr indexes are called **collections**, which is the sharded version of **cores**.

4.4.8.1 Zookeeper instances

Creating a SolrCloud connector is the same as creating a *Solr connector* with the only difference in the syntax of the `solrUrl` parameter:

```
"solrUrl":"zk://localhost:2181|numShards=2|replicationFactor=2|maxShardsPerNode=3"
```

`zk://localhost:2181` is the host and port of the started Zookeeper instance and the rest are the parameters for creating the SolrCloud collection, delimited with pipes. The supported cluster parameters are:

- `numShards`
- `replicationFactor`
- `maxShardsPerNode`
- `autoAddReplicas`
- `router.name`
- `router.field`
- `shards`

Note: `numShards` and `replicationFactor` are mandatory parameters. `maxShardsPerNode` is set to the `numShards` value when absent.

For more information on how to use these options, check the SolrCloud's Collection API documentation.

You can also have multiple Zookeeper instances orchestrating the Solr nodes. They must be named in the connection string.

```
"solrUrl":"zk://localhost:2181,zk://localhost:2182|numShards=2|replicationFactor=2|maxShardsPerNode=3"
```

Note: The Zookeeper instances must be running on the same hosts as in the `solrUrl` parameter.

More information on how to setup a SolrCloud cluster.

4.4.8.2 SolrCloud collection configsets

Unlike the standard Solr cores, where each core has a `/conf` directory containing all of its configurations, SolrCloud collections decouple the configuration from the data. The configurations are called **configsets** and they reside in the Zookeeper instances. Before you create a new collection, you must upload all your default or custom configurations to Zookeeper under specific names.

Note: Check Command Line Utilities and ConfigSets API from SolrCloud documentation on how to upload configsets.

When creating a SolrCloud connector, you must specify the configset name in the `copyConfigsFrom` parameter. If you do not, it will search for a default configset name, which is **collection1**. As a good practice, it is recommended

to upload your default configuration under the name **collection1**, and then, when you want to create a new connector with default index configuration, you will not have to specify this parameter again. Otherwise, for other custom configsets, use the parameter with the name of the custom configset such as **customConfigset**.

Example: Create SolrCloud connector query using a custom configset

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
    solr-index:my_collection :createConnector '''
{
  "solrUrl": "zk://localhost:2181|numShards=2|replicationFactor=2|maxShardsPerNode=3",
  "copyConfigsFrom": "customConfigset"
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "multivalued": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ]
    }
  ]
}
''' .
}
```

4.4.9 Caveats

4.4.9.1 Order of control

Even though SPARQL per se is not sensitive to the order of triple patterns, the Solr GraphDB Connector expects to receive certain predicates before others so that queries can be executed properly. In particular, predicates that specify the query or query options must come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.

4.4.10 Upgrading from previous versions

4.4.10.1 Migrating from GraphDB 9.x

GraphDB 10.0 introduces major changes to the filtering mechanism of the connectors. Existing connector instances will not be usable and attempting to use them for queries or updates will throw an error.

If your GraphDB 9.x (or older) connector definitions **do not** include an entity filter, you can simply *repair them*.

If your GraphDB 9.x (or older) connector definitions **do** include an entity filter with the `entityFilter` option, you need to rewrite the filter with one of the current *filter types*:

1. Save your existing connector definition.
2. Drop the connector instance.
3. In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter. Rewrite the filters as follows.

   Rule of thumb:

   - If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

   So if we take the example:

   ```
   ?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
   ```

   It needs to be rewritten like this:

   - Per-field rule on field `location`: `$this = <urn:Europe>`
   - Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
   - Top-level document filter: `BOUND(?location)`
4. Recreate the connector instance using the new definition.

4.5 Elasticsearch GraphDB Connector

Note: This feature requires a *GraphDB Enterprise license*.

Note: GraphDB supports *full-text search* options as well.

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Elasticsearch but with the additional benefit of staying automatically up to date with the GraphDB repository data.

The Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

Elasticsearch also allows for *vector search*, which uses semantic similarity to ensure better recall than full-text search. The embeddings used for this vector search can either be created by Elasticsearch, or by a client for an external embedding model configured in the connector. One of the options for embedding models is the default local model deployed with the Graphwise Transformer tool. You can learn more about configuring such a model under *Embedding model configuration for vector search*.

The main features of the GraphDB Connectors are:

- maintenance of an index that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entity synchronization defined by:
  - a list of fields (on the Elasticsearch side) and property chains (on the GraphDB side) whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization (the default is all languages)
  - additional filtering by property and value
- full-text search using native Elasticsearch queries
- snippet extraction: highlighting of search terms in the search result
- faceted search
- sorting by any preconfigured field
- paging of results using `OFFSET` and `LIMIT`
- custom mapping of RDF types to Elasticsearch types

Each feature is described in detail below.

See also:

Embedding model configuration for vector search

4.5.1 Usage

All interactions with the Elasticsearch GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the Elasticsearch GraphDB Connector, this is `http://www.ontotext.com/connectors/elasticsearch#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/elasticsearch#createConnector`) to create a connector instance for Elasticsearch.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For Elasticsearch, the instance IRI is `http://www.ontotext.com/connectors/elasticsearch/instance#`.

Warning: Deleting the repository does not remove the indexes in Elasticsearch. In order to create a connector instance with the same name as one of the instances in the deleted repository, you may need to delete the corresponding Elasticsearch index first.

Warning: Changing the Elasticsearch URL will not reindex the data automatically. If you need the data to be reindexed, you can *repair* the connector instance.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito, and Rozova, as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

4.5.2 Setup and maintenance

4.5.2.1 Prerequisites

Third-party component versions

The Elasticsearch Connector in GraphDB 11.3 supports Elasticsearch version 9.1, including any of the patch releases for that version.

Prerequisites for vector search

- Elasticsearch Enterprise is required if you want to Elasticsearch to create the embeddings when using vector search — check Elasticsearch's available subscription options for more details.
- When using another embedding model to generate the embeddings, you need to *configure the embedding model* before creating the connector.

4.5.2.2 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- an Elasticsearch instance to synchronize to
- classes to synchronize
- properties to synchronize

The configuration data must be provided as a JSON string representation and passed together with the create command.

You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill out the configuration form.

   If you are using the *Wine example dataset*, you can provide the following configuration parameters:

   - *Name*: `my_index`
   - *Fields*:
     - *Field name*: `grape`
     - *Property chain*:
       * `http://www.example.com/demo/wine#madeFromGrape`
       * `http://www.w3.org/2000/01/rdf-schema#label`
     - *Nested object fields*: Select *Indexed*, *Stored*, *Analyzed* and *Multivalued*
   - *Types*: `http://www.example.com/demo/wine#Wine`
   - *ES node*: `http://localhost:9200` (or the address of your Elasticsearch instance)

- Select *ES cluster sniff*, *ES manage index* and *ES manage mappings*
- *ES bulk update batch size*: `5000`
- *ES index create settings*: `5242880`

4. If you want to use vector search with your connector, you need to configure the fields for similarity search.
   1. If you want the embeddings to be created by GraphDB, provide the *embeddings model you want to use*.

      Tip: If you want Elasticsearch to create the embeddings, you do not need to change the embeddings model parameter.

   2. Specify the datatype of the field to use for similarity search. The datatype depends on whether the embeddings are created by Elasticsearch or by an embedding model via the connector, as described in the *documentation for the datatype field* below.
5. Execute the `CREATE` statement from the form by clicking *OK*. Alternatively, you can view its SPARQL query by clicking *View SPARQL Query*, and then copy it to execute it manually or integrate it in automation scripts.

   See also:

   List of creation parameters provides thorough explanation of all creation parameters.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `elastic:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
    elastic-index:my_index elastic:createConnector '''
{
  "elasticsearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
```

(continues on next page)

(continued from previous page)

```
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ]
}
''' .
}
```

The above command creates a new Elasticsearch connector instance that connects to the Elasticsearch instance accessible at port `9200` on the localhost as specified by the `elasticsearchNode` key.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to Elasticsearch. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: "grape", "sugar", and "year". The field names are used later in the queries.

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine's `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to the wine.

The fields `sugar` and `year` contain discrete values such as *medium*, *dry*, *2012*, and *2013*, so it is best to specify the option `analyzed: false` as well. See `analyzed` in *Defining fields* for more information.

Mapping and index management

By default, GraphDB manages (creates, deletes, or updates if needed) the Elasticsearch index and the Elasticsearch mapping. This makes it easier to use Elasticsearch as everything is done automatically. This behavior can be changed by the following options:

- `manageIndex`: if `true`, GraphDB manages the index. `true` by default.
- `manageMapping`: if `true`, GraphDB manages the mapping. `true` by default.

Note: If either of the options is set to `false`, you have to create, update or remove the `index/mapping` and, in case Elasticsearch is misconfigured, the connector instance will not function correctly.

Using a non-managed schema

The present version provides no support for changing some advanced options, such as stop words, on a per-field basis. The recommended way to do this for now is to manage the mapping yourself and tell the connector to just sync the object values in the appropriate fields. Here is an example:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
    elastic-index:my_index elastic:createConnector '''
{
  "elasticsearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
```

(continues on next page)

(continued from previous page)

```
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ],
  "manageMapping": false
}
''' .
}
```

This creates the same connector instance as above but it expects fields with the specified field names to be already present in the index mapping, as well as some internal GraphDB fields. For the example, you must have the following fields:

field name	Elasticsearch config
`_graphdb_id`	“type”:”long”, “index”:true, “store”:tru
`grape`	“type”:”text”, “index”:true, “store”:true
`sugar`	“type”:”keyword”, “index”:true, “store”:true
`year`	“type”:”keyword”, “index”:true, “store”:true

`_graphdb_id` is used internally by GraphDB and is always required.

Working with secured Elasticsearch

GraphDB allows the access of a secured Elasticsearch instance by passing the arbitrary `elasticsearchBasicAuthUser` and `elasticsearchBasicAuthPassword` parameters.

Instead of supplying the username and password as part of the connector instance configuration, you can also implement a custom authenticator class and set it via the `authenticationConfiguratorClass` option. See these connector authenticator examples for more information and example projects that implement such a custom class.

See the *List of creation parameters* for more information.

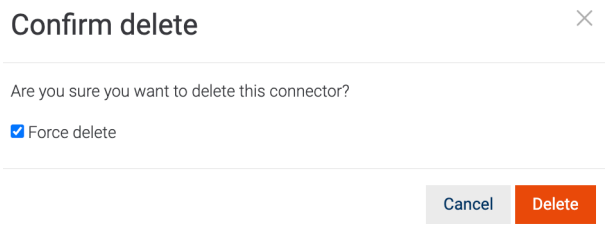
4.5.2.3 Dropping a connector instance

Dropping a connector instance removes all references to its external store from GraphDB as well as the Elasticsearch index associated with it.

The drop command is triggered by a SPARQL `INSERT` with the `elastic:dropConnector` predicate where the name of the connector instance must be in the subject position. For example, the following removes the connector `my_index`:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon 🗑, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.5.2.4 Retrieving the create options for a connector instance

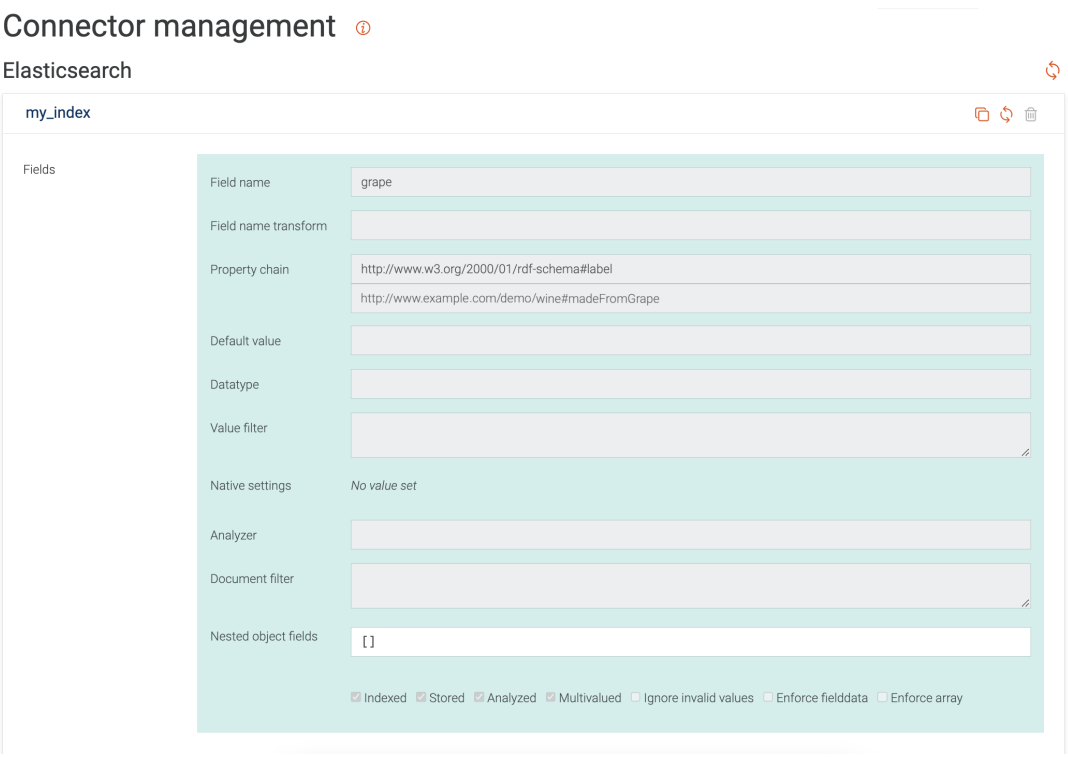
You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?createString {
  elastic-index:my_index elastic:listOptionValues ?createString .
}
```

4.5.2.5 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* 🔄 or *Delete* 🗑 icons to perform these operations. Click the *Copy* 📋 icon to copy the connector definition query to your clipboard.



With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?cntUri ?cntStr {
  ?cntUri elastic:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/elasticsearch/instance#my_index`, while `?cntStr` is bound to a string representing the local name such as "my_index" in this example.

4.5.2.6 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?cntUri ?cntStatus {
  ?cntUri elastic:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.5.3 Working with data

4.5.3.1 Adding, updating and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` update requests. This is achieved by intercepting all changes in the plugin and determining which Elasticsearch documents need to be updated.

4.5.3.2 Simple queries

Once a connector instance has been created, you can use SPARQL to query data from it. For each matching Elasticsearch document, the connector instance returns the document subject. In its simplest form, you query that data using a `SELECT` query that includes an Elasticsearch query as the object of the `elastic:query` predicate:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity {
  ?search a elastic-index:my_index ;
      elastic:query "grape:cabernet" ;
      elastic:entities ?entity .
}
```

The result of this query binds `?entity` to the two wines made from grapes that have “cabernet” in their name: `:Yoyowine` and `:Franvino`.

Note: You must use the field names you chose when you created the connector instance. They can be identical to the property IRIs but you must escape any special characters according to what Elasticsearch expects.

The creation of such a query follows these basic steps:

1. Get a query instance of the requested connector instance by using the RDF notation `"X a Y" (= X rdf:type Y)`, where `X` is a variable and `Y` is a connector instance IRI. `X` is bound to a query instance of the connector instance.
2. Assign a query to the query instance by using the system predicate `elastic:query`.
3. Request the matching entities through the `elastic:entities` predicate.

You can also provide per-query search options by using one or more option predicates. The option predicates are described in detail below.

4.5.3.3 Raw queries

To access an Elasticsearch query parameter that is not exposed through a special predicate, use a raw query. Instead of providing a full-text query in the `:query` part, specify raw Elasticsearch parameters. For example, to boost some parts of your full-text query as described in the Elastic documentation, execute the following query:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity {
  ?search a elastic-index:my_index ;
      elastic:query '''
      {
        "query" : {
          "bool" : {
```

(continues on next page)

(continued from previous page)

```
            "should" : [ {
              "query_string" : {
                "query" : "<full-text-query-not-boosted>"
              }
            }, {
              "query_string" : {
                "query" : "<full-text-query-boosted>",
                "boost" : 4.0
              }
            } ]
          }
        }
      }
      ''' ;
      elastic:entities ?entity .
}
```

Combining Elasticsearch results with GraphDB data

You can use the bound `?entity` variable in other SPARQL triple patterns to build complex queries that join to or fetch additional data from GraphDB. For example, to see the actual grapes in the matching wines as well as the year they were made:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?entity ?grape ?year {
  ?search a elastic-index:my_index ;
      elastic:query "grape:cabernet" ;
      elastic:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
  ?entity wine:hasYear ?year
}
```

The result looks like this:

	entity	grape	year
1	wine:Yoyowine	wine:CabernetSauvignon	"2013"^^xsd:integer
2	wine:Franvino	wine:Merlot	"2012"^^xsd:integer
3	wine:Franvino	wine:CabernetFranc	"2012"^^xsd:integer

Note: `:Franvino` is returned twice because it is made from two different grapes, both of which are returned.

Entity match score

It is possible to access the match score returned by Elasticsearch with the `score` predicate. As each entity has its own score, the predicate should come at the entity level. For example:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity ?score {
  ?search a elastic-index:my_index ;
      elastic:query "grape:cabernet" ;
      elastic:entities ?entity .
  ?entity elastic:score ?score
}
```

The result looks like this but the actual score might be different as it depends on the specific Elasticsearch version:

	entity	score
1	wine:Yoyowine	"0.3979403078556061"^^xsd:double
2	wine:Franvino	"0.33036553859710693"^^xsd:double

4.5.3.4 Vector search queries

You can conduct vector searches in Elasticsearch by using the similarity predicate in SPARQL or with raw Knn (K nearest neighbors) queries. However, querying data through raw Knn queries differs based on whether the embeddings are generated by Elasticsearch or *through another embedding model via the connector*.

Vector search using the similarity predicate

You can use SPARQL to query data using the `similarity` predicate. The queries are the same regardless of what creates the embeddings.

```
PREFIX conn:<http://www.ontotext.com/connectors/elasticsearch#>
PREFIX inst:<http://www.ontotext.com/connectors/elasticsearch/instance#>


SELECT ?entity WHERE {
   ?search a inst:my-index ;
          conn:similarity "myvectorfield:mysearchterm1 mysearchterm2" ;
          conn:entities ?entity .
}
```

Vector search using raw Knn queries

You can also use raw Knn queries to access data not exposed through a special predicate.

Raw query with embeddings created by an embedding model:

```
PREFIX conn:<http://www.ontotext.com/connectors/elasticsearch#>
PREFIX inst:<http://www.ontotext.com/connectors/elasticsearch/instance#>


SELECT ?entity WHERE {
   ?search a inst:my-index ;
```

(continues on next page)

(continued from previous page)

```
          conn:query '''{
                      "knn": {
                        "field": "myvectorfield",
                        "query_vector": "mysearchterm1 mysearchterm2",
                        "k": 100,
                        "num_candidates": 150
                      }
                  }''' ;
          conn:entities ?entity .
}
```

You can also combine Knn vector search queries and full text search in a raw query:

```
PREFIX conn:<http://www.ontotext.com/connectors/elasticsearch#>
PREFIX inst:<http://www.ontotext.com/connectors/elasticsearch/instance#>


SELECT ?entity WHERE {
   ?search a inst:my-index ;
conn:query '''{
                    "query": {
                      "bool": {
                        "must": [
                          {
                            "match": {
                              "mytextfield": {
                                "query": "mysearchterm",
                                "boost": 1
                              }
                            }
                          },
                          {
                            "knn": {
                              "field": "myvectorfield",
                              "query_vector": "mysearchterm1 mysearchterm2",
                              "k": 10,
                              "boost": 2
                            }
                          }
                        ]
                      }
                    }
                }''' ;
          conn:entities ?entity .
}
```

Raw query with embeddings created by Elasticsearch:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>


SELECT ?entity ?name ?abstract {
  ?search a elastic-index:my-index;
      elastic:query '''{
                          "query": {
                            "semantic": {
                              "field": "mysemantictextfield",
                              "query": "mysearchterm"
                            }
                          }
```

(continues on next page)

(continued from previous page)

```
                        }''' ;
      elastic:entities ?entity .
    ?entity <http://schema.org/name> ?name.
    ?entity <http://schema.org/abstract> ?abstract .
}
```

Vector search in nested objects

If you have nested objects with "`vector`" or "`native:semantic_text`" fields, you can also execute vector queries that will search in the nested objects. To do so, you need to define the query as a nested query and provide the full path to the object in the query.

See also:

The tutorial on *Vector search in nested fields with Elasticsearch* provides an in-depth example on how to create an Elasticsearch connector instance with vector fields in nested objects and execute queries, including multi-level nested objects.

4.5.3.5 Basic facet queries

You can also query facets with your connector instance. For example, with the sample `my_index` connector described above:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?facetName ?facetValue ?facetCount WHERE {
  # note empty query is allowed and will just match all documents, hence no elastic:query
  ?r a elastic-index:my_index ;
    elastic:facetFields "year,sugar" ;
    elastic:facets [
    elastic:facetName ?facetName;
    elastic:facetValue ?facetValue;
    elastic:facetCount ?facetCount
  ]
}
```

It is important to specify the facet fields by using the `facetFields` predicate. Its value is a simple comma-delimited list of field names. In order to get the faceted results, use the `elastic:facets` predicate. Because each facet has three components (`name`, `value`, and `count`), the `elastic:facets` predicate returns multiple nodes that can be used to access the individual values for each component through the predicates `facetName`, `facetValue`, and `facetCount`.

The resulting bindings will look like this:

	facetName	facetValue	facetCount
1	"year"	"2012"	"3"^^xsd:long
2	"year"	"2013"	"2"^^xsd:long
3	"sugar"	"dry"	"3"^^xsd:long
4	"sugar"	"medium"	"2"^^xsd:long

You can easily see that there are three wines produced in 2012 and two in 2013. You also see that three of the wines are dry, while two are medium. However, it is not necessarily true that the three wines produced in 2012 are the same as the three dry wines as each facet is computed independently.

Tip: Faceting by analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens, and faceting uses each token to create a faceting bucket. For example, "North America" and "Europe" produce three buckets: "north", "america", and "europe", corresponding to each token in the two values.

If you need to facet by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.5.3.6 Advanced facet and aggregation queries

While basic faceting allows for simple counting of documents based on the discrete values of a particular field, there are more complex faceted or aggregation searches in Elasticsearch. The Elasticsearch GraphDB Connector provides a mapping from Elasticsearch results to RDF results but no mechanism for specifying the queries other than executing *Raw queries*.

Supported Elasticsearch facets and aggregations

The Elasticsearch GraphDB Connector supports mapping of the following facets and aggregations:

- Facets: terms, histogram, date histogram;
- Aggregations: terms, histogram, date histogram, range, min, max, sum, avg, stats, extended stats, value count.

For aggregations, the connector also supports sub-aggregations.

Tip: For more information on each supported facet or aggregation type, refer to the Elasticsearch documentation.

RDF mapping of the results

The results are accessed through the predicate `aggregations` (much like the basic facets are accessed through `facets`). The predicate binds multiple blank nodes that each contains a single aggregation bucket. The individual bucket items can be accessed through these predicates:

Predicate	Meaning	Elasticsearch counterpart
`:name`	Bucket name	`getName()`
`:key`	Key or value associated with the bucket	`getValue()` or `getKey()`
`:count`	Count of documents in the bucket	`getDocCount()`, `getValue()`
`:from`	Start of range	`getFrom()`, `getFromAsDate()`
`:to`	End of range (RangeFacet)	`getTo()`, `getToAsDate()`
`:min`	Minimum value	`getMin()`, `getValue()`
`:max`	Maximum value	`getMax()`, `getValue()`
`:sum`	Sum value	`getSum()`, `getValue()`
`:avg`	Average value	`getAvg()`, `getValue()`
`:sum_of_squares`	Sum of squares value	`getSumOfSquares()`
`:variance`	Variance value	`getVariance()`
`:std_deviation`	Standard deviation value	`getStdDeviation()`
`:parent`	Sub-aggregations: points to the parent (upper level) blank node	
`:level`	Sub-aggregations: level number where 1 is the uppermost level and the following levels are 2, 3 and so on	
`:levelName`	Sub-aggregations: level name	`getKey()` or `getValue()`

4.5.3.7 Sorting

You can sort the entities returned by a connector query according to one or more fields. To do this, assign a comma-delimited list of sort fields to the `orderBy` predicate. Each field can be prefixed with a minus to indicate sorting in descending order. For example:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>
SELECT ?entity ?sugar{
  ?search a elastic-index:my_index ;
      elastic:query "year:2013" ;
      elastic:orderBy "-sugar" ;
      elastic:entities ?entity.
  ?entity wine:hasSugar ?sugar
}
```

The result of this sample query contains wines produced in 2013 sorted according to their sugar content in descending order:

	entity	sugar
1	wine:Rozova	"medium"
2	wine:Yoyowine	"dry"

By default, entities are sorted according to their matching score in descending order.

Note: If you join the entity from the connector query to other triples stored in GraphDB, GraphDB might scramble the order. Avoid this by using SPARQL's `ORDER BY` instruction.

Tip: Sorting by an analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens and sorting uses the least (in the lexicographical sense) token. For example, "North America" will be sorted before "Europe" because the token "america" is lexicographically smaller than the token "europe". If you need to sort by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.5.3.8 Limit and offset

Limit and offset are supported on the Elasticsearch side of the query. You can set these values using the predicates `limit` and `offset`. Consider this example in which an offset of `1` and a limit of `1` are specified:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity {
  ?search a elastic-index:my_index ;
      elastic:query "sugar:dry" ;
      elastic:offset "1" ;
      elastic:limit "1" ;
      elastic:entities ?entity .
}
```

`offset` is counted from 0. The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:

	entity
1	wine:Yoyowine
2	wine:Franvino
3	wine:Blanquito

Note: The specific order in which GraphDB returns the results depends on how Elasticsearch returns the matches, unless sorting is specified.

4.5.3.9 Snippet extraction

Snippet extraction lets you retrieve highlighted snippets of text that match a query. You can access the snippets by using the the dedicated predicate `elastic:snippets`. This binds a blank node that in turn provides the actual snippets via the predicates `elastic:snippetField` and `elastic:snippetText`. The predicate snippets must be attached to the entity, as each entity has a different set of snippets. For example, in a search for Cabernet:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a elastic-index:my_index ;
      elastic:query "grape:cabernet" ;
      elastic:entities ?entity .
  ?entity elastic:snippets ?snippet .
  ?snippet elastic:snippetField ?snippetField ;
     elastic:snippetText ?snippetText .
}
```

the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

	entity	snippetField	snippetText
1	wine:Yoyowine	"grape"	"Cabernet Sauvignon"
2	wine:Franvino	"grape"	"MerlotCabernet Franc"

Note: The actual snippets might be different depending on the specific Elasticsearch implementation.

You can tweak how the snippets are collected and composed by using the following option predicates:

- `elastic:snippetSize`: sets the maximum size of the extracted text fragment, `250` by default
- `elastic:snippetSpanOpen`: the text to insert before the highlighted text, `<em>` by default
- `elastic:snippetSpanClose`: the text to insert after the highlighted text, `</em>` by default

The option predicates are set on the query instance, much like the `elastic:query` predicate.

Snippets from nested documents

Note: This section uses the *Nested objects* dataset described later in this page.

Snippets extracted from nested documents (when a nested query is used) will be available through the same mechanism as snippets from non-nested fields. In addition, nested snippet results provide the nested search path via the `snippetInnerField` predicate. For example, in a nested search on the field "grandChildren" (specified by "path") and a match query for "tylor" on the nested field "grandChildren.name":

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity ?snippetInnerField ?snippetField ?snippetText {
  ?search a elastic-index:my_index ;
      elastic:query '''
        {
           "query":{
              "nested":{
                 "path":"grandChildren",
                 "query":{
                    "bool":{
                       "must":[
                          {
                             "match":{
                                "grandChildren.name":"tylor"
                             }
                          }
                       ]
                    }
                 }
              }
           }
        }
      ''' ;
      elastic:entities ?entity .
  ?entity elastic:snippets ?snippet .
  ?snippet elastic:snippetInnerField ?snippetInnerField ;
           elastic:snippetField ?snippetField ;
           elastic:snippetText ?snippetText .
}
```

the query returns all people who have a grandchild whose name matches "tylor", as well as the highlighted snippets:

?entity	?snippetInnerField	?snippetField	?snippetText
urn:Eva	grandChildren	grandChildren.name	John-Tylor
urn:John	grandChildren	grandChildren.name	John-Tylor
urn:John	grandChildren	grandChildren.name	Tylor
urn:Mary	grandChildren	grandChildren.name	Tylor

Note that the matching field whose matching values are highlighted is provided via the `snippetField` predicate, just like extracting snippets with non-nested searches, while the predicate `snippetInnerField` provides the field on which the nested search was executed.

4.5.3.10 Total hits

You can get the total number of matching Elasticsearch documents (hits) by using the `elastic:totalHits` predicate. For the `my_index` connector instance used above, a query for all wines made in 2012 might look like this:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?totalHits {
    ?r a elastic-index:my_index ;
       elastic:query "year:2012" ;
       elastic:totalHits ?totalHits .
}
```

As there are three wines made in 2012, the value `3` (of type `xsd:long`) binds to `?totalHits`.

As you see above, you can omit the return of any of the matching entities. This can be useful if there are many hits and you want to calculate pagination parameters.

4.5.4 List of creation parameters

The creation parameters define how the `elastic:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object, where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the *Create Connector user interface* without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal internal data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to make an Elasticsearch index from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `elastic:graph`, where the prefix `elastic:` represents the IRI `http://www.ontotext.com/connectors/elasticsearch#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *read-only mode*.

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
INSERT {
    GRAPH elastic:graph {
        ...
    }
} WHERE {
        ...
};
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
INSERT DATA {
```

(continues on next page)

(continued from previous page)

```
    elastic-index:my_index elastic:createConnector '''
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponding to a single predicate with a field name the same as the predicate, so you must use escaping when issuing Elasticsearch queries.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`elasticsearchNode` (string), required, the Elasticsearch instance to sync to As Elasticsearch is a third-party service, you have to specify the node where it is running. The node value takes the form `http://hostname.domain:port` or `https://hostname.domain:port`. No default value. Can be updated at runtime with no need to rebuild the index.

> **Note:** Elasticsearch exposes two protocols — the native transport* protocol over port `9300` and the RESTful API over port `9200`. The Elasticsearch GraphDB Connector uses the RESTful API over port `9200`.

`indexCreateSettings` (json), optional, the settings for creating the Elasticsearch index This option is passed directly to Elasticsearch when creating the index.

`elasticsearchBasicAuthUser` (string), optional, the settings for supplying the authentication user No default value. Can be updated at runtime with no need to rebuild the index.

`elasticsearchBasicAuthPassword` (string), optional, the settings for supplying the authentication password A password is a string with a single value that is not logged or printed. No default value. Can be updated at runtime with no need to rebuild the index.

`elasticsearchClusterSniff` (boolean), controls whether to build the server address list by sniffing on the Elasticsearch clus Corresponds to the Elasticsearch `client.transport.sniff` option. `true` by default Can be updated at runtime with no need to rebuild the index.

`bulkUpdateBatchSize` (integer), controls the maximum number of documents sent per bulk request Default value is `5,000`. Can be updated at runtime with no need to rebuild the index.

`authenticationConfiguratorClass` provides custom authentication behavior Optional.

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type. See also the examples in *General full-text search with the connectors*.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual, but only some of the languages represented in the literal values can be mapped. You can do this by specifying a list of language ranges to be matched to the language tags of literals according to RFC 4647, Section 3.3.1. Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to Elasticsearch The fields specify exactly which parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in Elasticsearch. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

- **`fieldName` (string), required, the name of the field in Elasticsearch** The name of the field defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field name is used at query time to refer to the field. There are few restrictions on the allowed characters in a field name, but to avoid unnecessary escaping (which depends on how Elasticsearch parses its queries), we recommend keeping the field names simple.
- **`fieldNameTransform` (one of `none`, `predicate` or `predicate.localName`), optional, `none` by default** Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

  See *Indexing all literals in distinct fields* for an example.
- **`propertyChain` (list of IRI), required, defines the property chain to reach the value** The property chain defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple, etc. In this model, a property chain with a single element corresponds to a direct property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

  The IRI of the document will be synchronized to the special field `_id` in Elasticsearch. You may use it to query Elasticsearch directly and to retrieve the matching entity IRI.

  See *Copy fields* for defining multiple fields with the same property chain.

  See *Multiple property chains per field* for defining a field whose values are populated from more than one property chain.

  See *Indexing language tags* for defining a field whose values are populated with the language tags of literals.

  See *Indexing the IRI of an entity* for defining a field whose values are populated with the IRI of the indexed entity.

  See *Wildcard literal indexing* for defining a field whose values are populated with literals regardless of their predicate.
- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.
- **`documentFilter` (string), optional, specifies the nested document filter for the field** Only for fields that define nested documents. See also *Entity filtering*.
- **`defaultValue` (string), optional, specifies a default value for the field** Provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or a IRI. It has no default value.

- **`indexed` (boolean), optional, default `true`** If indexed, a field is available for Elasticsearch queries. `true` by default.

  If `true`, this option corresponds to `"index" = true`. If `false`, it corresponds to `"index" = false`.

- **`stored` (boolean), optional, default `true`** Whether fields are stored in Elasticsearch. `true` by default.

  This option corresponds to the property `"store"` in the Elasticsearch mapping.

- **`analyzed` (boolean), optional, default `true`** When literal fields are indexed in Elasticsearch, they will be analyzed according to the analyzer settings. Should you require that a given field is not analyzed, you may use `"analyzed"`. This option has no effect for IRIs, which are never analyzed. `true` by default.

  If `true`, this option will use automatic or manual (`datatype` option) type for the Elasticsearch mapping. If `false`, it corresponds to `"type" = "keyword"` (i.e., the default type will be changed to keyword).

- **`multivalued` (boolean), optional, default `true`** RDF properties and synchronized fields may have more than one value. If `multivalued` is set to `true`, all values will be synchronized to Elasticsearch. If set to `false`, only a single value will be synchronized. `true` by default.

- **`ignoreInvalidValues` (boolean), optional, default `false`** Per-field option that controls what happens when a value cannot be converted to the requested (or previously detected) type. `False` by default.

  Example use: when an invalid date literal like `"2021-02-29"^^xsd:date` (2021 is not a leap year) needs to be indexed as a date, or when an IRI needs to be indexed as a number.

  > **Note:** Some conversions are always valid: any literal to an FTS field, any non-literal (IRI, blank node, embedded triple) to a non-analyzed field. When `true`, such values will be skipped with a note in the logs. When `false`, such values will break the transaction.

- **`array` (boolean), optional, default `false`** Normally, Elasticsearch creates an array only if more than value is present for a given field. If `array` is set to `true`, Elasticsearch will always create an array even for single values. If set to `false`, Elasticsearch will create arrays for multiple values only. `False` by default.

- **`fielddata` (boolean), optional, default `false`** Allows fielddata to be built in memory for text fields. This can consume a lot of heap space, especially when loading high cardinality text fields. `False` by default.

- **`datatype` (string), optional, the manual datatype override** By default, the Elasticsearch GraphDB Connector uses datatype of literal values to determine how they should be mapped to Elasticsearch types. For more information on the supported datatypes, see *Datatype mapping*.

  You can override the datatype mapping with the `"datatype"` parameter, which can be specified per field. The value of `datatype` can be any of the `xsd:` types supported by the automatic mapping or a native Elasticsearch type prefixed by `native:`. For example, both `xsd:long` and `native:long` map to the long type in Elasticsearch.

  When using vector similarity search, you need to provide the datatype of the field that will be used for similarity search:

  - If the embeddings are created by an externally provided model (such as a locally deployed model or an OpenAI model) via GraphDB, specify `"vector"` as the value of the `"datatype"` parameter. This datatype creates a `dense_vector` field.
  - If the embeddings are created by Elasticsearch, specify `"native:semantic_text"`.

    > **Note:** If you want to use an inference pipeline from Elasticsearch, specify `"nativeSettings": {"inference_id": "your_inference_id"}`.

You can also create a single vector from multiple fields by specifying `"vector"` or `"native:semantic_text"` (depending on *where the embeddings are generated*) as the datatype for each of the fields you want to use.

- **`nativeSettings` (json), optional, custom field settings** The setting for the Elasticsearch mapping parameters of the respective field such as the `format` of the datatype. Native field settings require an explicit native datatype.

  `nativeSettings` are not allowed for the following parameters so as to avoid conflicts with the existing way to specify them: `type`, `index`, `store`, `analyzer`, `fielddata`.

- **`objectFields` (objects array), optional, nested object mapping** When `native:object`, `native:nested`, or `native:geo_point` is used as a datatype value, this provides a mapping for the nested object's fields. If `datatype` is not provided, then `native:object` will be assumed.

  For the difference between `object` and `nested`, refer to the Elastic nested field type. The `geo_point` type must have exactly two fields named `lat` and `long` (required by Elastic, see geo-point field type).

  Nested objects support further nested objects with a limit of five levels of nesting. See *Nested objects* for an example.

- **`startFromParent` (integer), optional, default `0`** Start processing the property chain from the N-th parent instead of the root of the current nested object. 0 is the root of the current nested object, 1 is the parent of the nested object, 2 is the parent of the parent and so on.

- **`analyzer` (string), optional, per field analyzer** The Elasticsearch analyzer that is used for indexing the field can be specified with the parameter `analyzer`. It will be passed directly to Elasticsearch's property `analyzer` when creating the mapping (see Custom Analyzers in the Elasticsearch documentation). For example:

```
{
...
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzer": "my_analyzer"
    },
...
}
```

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

`embeddingModel` (string), optional, specifies the embedding model used to create the embeddings when using vector search See also *Embedding model configuration for vector search*

4.5.4.1 Updating parameters at runtime

As mentioned above, you can update the following connection parameters at runtime with no need to rebuild the index:

- `elasticsearchNode`
- `elasticsearchClusterSniff`
- `elasticsearchBasicAuthUser`
- `elasticsearchBasicAuthPassword`
- `bulkUpdateBatchSize`

The following SPARQL update request demonstrates this by updating the user and password values:

```
PREFIX conn:<http://www.ontotext.com/connectors/elasticsearch#>
PREFIX inst:<http://www.ontotext.com/connectors/elasticsearch/instance#>
INSERT DATA {
inst:proper_index conn:updateConnector '''
    {
        "elasticsearchBasicAuthUser": "foo",
        "elasticsearchBasicAuthPassword": "bar"
    }
''' .
}
```

4.5.4.2 Special field definitions

Nested objects

Nested objects are Elasticsearch documents that are used as values in the main document or in other nested objects. Up to five levels of nesting is possible. They are defined with the *objectFields* option.

Let's look at an example that uses the following data, which consists of children and grandchildren relations:

```
<urn:John>
    a <urn:Person> ;
    <urn:name> "John" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 60 ;
    <urn:hasSpouse> <urn:Mary> ;
    <urn:hasChild> <urn:Billy> ;
    <urn:hasChild> <urn:Annie> .

<urn:Mary>
    a <urn:Person> ;
    <urn:name> "Mary" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 58 ;
    <urn:hasSpouse> <urn:John> ;
    <urn:hasChild> <urn:Billy> .

<urn:Eva>
    a <urn:Person> ;
    <urn:name> "Eva" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 45 ;
    <urn:hasChild> <urn:Annie> .

<urn:Billy>
```

(continues on next page)

(continued from previous page)

```
    a <urn:Person> ;
    <urn:name> "Billy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 35 ;
    <urn:hasChild> <urn:Tylor> ;
    <urn:hasChild> <urn:Melody> .

<urn:Annie>
    a <urn:Person> ;
    <urn:name> "Annie" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 28 ;
    <urn:hasChild> <urn:Sammy> .

<urn:Tylor>
    a <urn:Person> ;
    <urn:name> "Tylor" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 5 .

<urn:Melody>
    a <urn:Person> ;
    <urn:name> "Melody" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 2 .

<urn:Sammy>
    a <urn:Person> ;
    <urn:name> "Sammy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 10 .

<urn:Male> <urn:label> "male" .

<urn:Female> <urn:label> "female" .
```

We can create a nested objects index that consists of children and grandchildren with their corresponding fields defining their gender and age. We use the `native:nested` type because we want to query the nested objects independently of each other:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["urn:Person"],
      "fields": [
        {
          "fieldName": "name",
          "propertyChain": [
            "urn:name"
          ]
        },
        {
          "fieldName": "age",
          "propertyChain": [
            "urn:age"
          ],
          "datatype": "xsd:long"
```

(continues on next page)

(continued from previous page)

```
      },
      {
        "fieldName": "hasSpouse",
        "propertyChain": [
          "urn:hasSpouse"
        ]
      },
      {
        "fieldName": "gender",
        "propertyChain": [
          "urn:gender",
          "urn:label"
        ]
      },
      {
        "fieldName": "children",
        "propertyChain": [
          "urn:hasChild"
        ],
        "datatype": "native:nested",
        "objectFields": [
          {
            "fieldName": "id",
            "propertyChain": [
              "$self"
            ]
          },
          {
            "fieldName": "name",
            "propertyChain": [
              "urn:name"
            ]
          },
          {
            "fieldName": "age",
            "propertyChain": [
              "urn:age"
            ],
            "datatype": "xsd:long"
          },
          {
            "fieldName": "gender",
            "propertyChain": [
              "urn:gender",
              "urn:label"
            ]
          },
          {
            "fieldName": "children",
            "propertyChain": [
              "urn:hasChild"
            ],
            "objectFields": [
              {
                "fieldName": "id",
                "propertyChain": [
                  "$self"
                ]
              },
              {
                "fieldName": "name",
```

(continues on next page)

(continued from previous page)

```
                "propertyChain": [
                  "urn:name"
                ]
              },
              {
                "fieldName": "age",
                "propertyChain": [
                  "urn:age"
                ],
                "datatype": "xsd:long"
              }
            ]
          }
        ]
      },
      {
        "fieldName": "grandChildren",
        "valueFilter": "$this -> type in (<urn:Person>)",
        "propertyChain": [
          "urn:hasChild",
          "urn:hasChild"
        ],
        "datatype": "native:nested",
        "objectFields": [
          {
            "fieldName": "id",
            "propertyChain": [
              "$self"
            ]
          },
          {
            "fieldName": "name",
            "propertyChain": [
              "urn:name"
            ]
          },
          {
            "fieldName": "age",
            "propertyChain": [
              "urn:age"
            ],
            "datatype": "xsd:long"
          },
          {
            "fieldName": "gender",
            "propertyChain": [
              "urn:gender",
              "urn:label"
            ]
          }
        ]
      },
    ]
  }
'''
}
```

To find male grandchildren of age 5 and older, we will use the following query:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
```

(continues on next page)

(continued from previous page)

```
SELECT ?entity {
    ?search a elastic-index:my_index ;
        elastic:query '''
            {
               "query" : {
                  "nested" : {
                     "path" : "grandChildren",
                     "query" : {
                        "bool" : {
                           "must" : [
                              {
                                 "match" : {
                                    "grandChildren.gender" : "male"
                                 }
                              },
                              {
                                 "range" : {
                                    "grandChildren.age" : {
                                       "gt" : 5
                                    }
                                 }
                              }
                           ]
                        }
                     }
                  }
               }
            }
        ''' ;
        elastic:entities ?entity .
}
ORDER BY ?entity
```

The result looks like this:

?entity
urn:Eva
urn:John

Nested objects with vector datatypes

You can also execute vector search queries with nested objects by adding fields to the nested object with "`vector`" or "`native:semantic_text` as the value of the "`datatype`" parameter.

See also:

- *The documentation for the datatype field below* for more information on which datatype to use
- *Vector search in nested objects*
- The tutorial on *Vector search in nested fields with Elasticsearch* for an in-depth example on how to create an Elasticsearch connector instance with vector fields in nested objects

Copy fields

It is often convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases such as faceting or sorting vs full-text search. The Elasticsearch GraphDB Connector has explicit support for fields that copy their value from another field. To do this, specify a single element in the property chain of the form `@otherFieldName`, where `otherFieldName` is another non-copy field. Take the following example:

```
...
  "fields": [
    {
      "fieldName": "grape",
      "facet": false,
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzed": true
    },
    {
      "fieldName": "grapeFacet",
      "propertyChain": [
        "@grape"
      ],
      "analyzed": false
    }
  ]
...
```

The snippet creates an analyzed field "grape" and a non-analyzed field "grapeFacet"; both fields are populated with the same values and "grapeFacet" is defined as a copy field that refers to the field "facet".

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in Elasticsearch) with more than one property chain — for example, the concept of "name" could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in Elasticsearch, you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.example.com/demo#canonicalName"
      ],
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.example.com/demo#historicalName"
      ]
      ...
```

(continues on next page)

(continued from previous page)

```
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in Elasticsearch.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old`, you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "lang()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.example.com/demo#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "graph()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.example.com/demo#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "feature",
          "propertyChain": [
            "http://www.example.com/demo#feature",
            "localName()"
          ]
        }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.example.com/demo#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
    elastic-index:my_index elastic:createConnector '''
{
  "elasticsearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "entityId",
```

(continues on next page)

(continued from previous page)

```
      "propertyChain": [
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

Note: Note that GraphDB will also use the IRI of each entity as the ID of each document in Elasticsearch, which is represented by the field `id`.

4.5.5 Datatype mapping

The Elasticsearch GraphDB Connector maps different types of RDF values to different types of Elasticsearch values according to the basic type of the RDF value (IRI or literal) and the datatype of literals. The auto-detection uses the following mapping:

RDF value	RDF datatype	Elasticsearch type
IRI	n/a	`keyword`
literal	any type not explicitly mentioned below	`text`
literal	`xsd:boolean`	`boolean`
literal	`xsd:double`	`double`
literal	`xsd:float`	`float`
literal	`xsd:long`	`long`
literal	`xsd:int`	`integer`
literal	`xsd:dateTime`	`date` with `format: strict_date_time`
literal	`xsd:date`	`date` with `format: strict_date`
literal	`xsd:time`	`date` with `format: strict_time_no_millis||strict_time`
literal	`xsd:gYear`	`date` with `format: strict_year`
literal	`xsd:gYearMonth`	`date` with `format: strict_year_month`

For any given field, the automatic mapping uses the first value it sees. This works fine for clean datasets but might lead to problems if your dataset has non-normalized data — for example, the first value has no datatype but other values do. Because of this, it's best to set the *datatype* to a fixed value such as `xsd:date`.

Note: There are two additional datatypes used for vector search but not by the datatype mapping mechanism — see the *datatype entry in the Creation parameters section* for more details on which values to use when creating an Elasticsearch connector instance with vector search capabilities.

Warning: The commonly used `xsd:integer` and `xsd:decimal` datatypes are not indexed as numbers because they represent infinite precision numbers. You can override that by using the *datatype* option to cast them to `xsd:long`, `xsd:double`, or `xsd:float` as appropriate.

4.5.5.1 Date and time conversion

RDF and Elasticsearch use slightly different models for representing dates and times, even though the values might look very similar.

Years in RDF values use the XSD format and are *era* years, where positive values denote the common era and negative values denote years before the common era. There is no year zero.

Years in Elasticsearch use the ISO 8601 format and are *proleptic* years: positive values denote years from the common era with any previous eras just going down by one mathematically so there is year zero.

In short:

- year 2020 CE = year 2020 in XSD = year 2020 in ISO.
- …
- year 1 CE = year 1 in XSD = year 1 in ISO.
- year 1 BCE = year -1 in XSD = year 0 in ISO.
- year 2 BCE = year -2 in XSD = year -1 in ISO.
- …

All years coming from RDF literals will be converted to ISO before indexing in Elasticsearch.

Both XSD and ISO date and time values support timezones. In addition to that, XSD defines the lack of a timezone as *undetermined*. Since we do not want to have any undetermined state in the indexing system, GraphDB defines the undetermined time zone as UTC. For example, `"2020-02-14T12:00:00"^^xsd:dateTime` is equivalent to `"2020-02-14T12:00:00Z"^^xsd:dateTime` (Z is the UTC timezone, also known as +00:00).

Also note that XSD dates and partial dates, e.g., `xsd:gYear` values, may have a timezone, which leads to additional complications. For example, `"2020+02:00"^^xsd:gYear` (the year 2020 in the +02:00 timezone) will be normalized to `2019-12-31T22:00:00Z` (the previous year!) if GraphDB followed strict timezone adherence. We have chosen to ignore the timezone on any values that do not have an associated time value. For example, all of the following values will be treated as if they specified UTC as their timezone:

- `"2020-02-15+02:00"^^xsd:date`
- `"2020-02+02:00"^^xsd:gYearMonth`
- `"2020+02:00"^^xsd:gYear`

4.5.6 Entity filtering

The Elasticsearch connector supports four kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to Elasticsearch if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable.

4.5.6.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

On nested documents, the per-field value filter can be used to remove the entire nested document early in the indexing process, e.g., by checking the type of the nested document via next hop with `rdf:type`.

Nested document filter The nested document filter is specified with the `documentFilter` parameter **inside the field definition** of the field that defines the root of a nested document. The filter is evaluated after the entire nested document has been collected. Failing to pass this filter removes the entire nested document.

Inside a nested document filter, the field names are within the context of the nested document and not within the context of the top-level document. For example, if you have a `children` field that defines a nested document and you use a filter like `?age < "10"^^xsd:int`, it will refer to the field `children.age`. You can use the prefix `$outer.` one or more times to refer to field values from the outer document from the viewpoint of the nested document. For example, `$outer.age > "25"^^xsd:int` will refer to the `age` field that is a sibling of the `children` field.1

Other than the above differences, the nested document filter is equivalent to the top-level document filter from the viewpoint of the nested document.

See also *Migrating from GraphDB 9.x*.

4.5.6.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on original RDF values before they are converted to Elasticsearch values using *datatype mapping*.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`

Continued on next page

Table 10 – continued from previous page

Operator	Meaning
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible such as `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height` has a value of `"150"^^xsd:int` and `dateOfBirth` has a
	value of `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

Continued on next page

Table 10 – continued from previous page

Operator	Meaning

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.5.6.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.5.6.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter and the nested document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.5.6.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.example.com/demo#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
```

(continues on next page)

(continued from previous page)

```
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": ["http://www.example.com/demo#name"]
         },
         {
           "fieldName": "city",
           "propertyChain": ["http://www.example.com/demo#city"],
           "valueFilter": "$this = \\"London\\""
         }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
         {
           "fieldName": "city",
           "propertyChain": ["http://www.example.com/demo#city"],
           "defaultValue": "London"
         }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.5.6.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.example.com/demo2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
```

(continues on next page)

(continued from previous page)

```
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
    rdf:type example2:Person ;
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to Elasticsearch, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  elastic-index:my_index elastic:createConnector '''
    {
      "elasticsearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "taggedWithPerson",
          "propertyChain": ["http://www.example.com/demo2#taggedWith"],
          "valueFilter": "$this -> type = <http://www.example.com/demo2#Person>"
        },
        {
          "fieldName": "taggedWithLocation",
          "propertyChain": ["http://www.example.com/demo2#taggedWith"],
          "valueFilter": "$this -> type = <http://www.example.com/demo2#Location>"
        }
      ]
    }
  ''' .
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

You can check this by issuing a faceted search for `taggedWithLocation` and `taggedWithPerson`:

```
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?facetName ?facetValue ?facetCount {
  ?search a elastic-index:my_index ;
      elastic:facetFields "taggedWithLocation,taggedWithPerson" ;
      elastic:facets [
        elastic:facetName ?facetName ;
        elastic:facetValue ?facetValue ;
        elastic:facetCount ?facetCount
      ]
}
```

If the filter was applied, you should get only `:Berlin` for `taggedWithLocation` and only `:Einstein` and `:Mozart` for `taggedWithPerson`:

facetName	facetValue	facetCount
taggedWithLocation	`http://www.example.com/demo2#Berlin`	3
taggedWithPerson	`http://www.example.com/demo2#Mozart`	2
taggedWithPerson	`http://www.example.com/demo2#Einstein`	1

4.5.6.7 Migrating filters from GraphDB 9.x

If you used entity filters in the connectors in GraphDB 9.x (or older) with the `entityFilter` option, you need to rewrite them using one of the current *filter types*.

In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter.

Rule of thumb:

- If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
- If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

So if we take the example:

```
?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
```

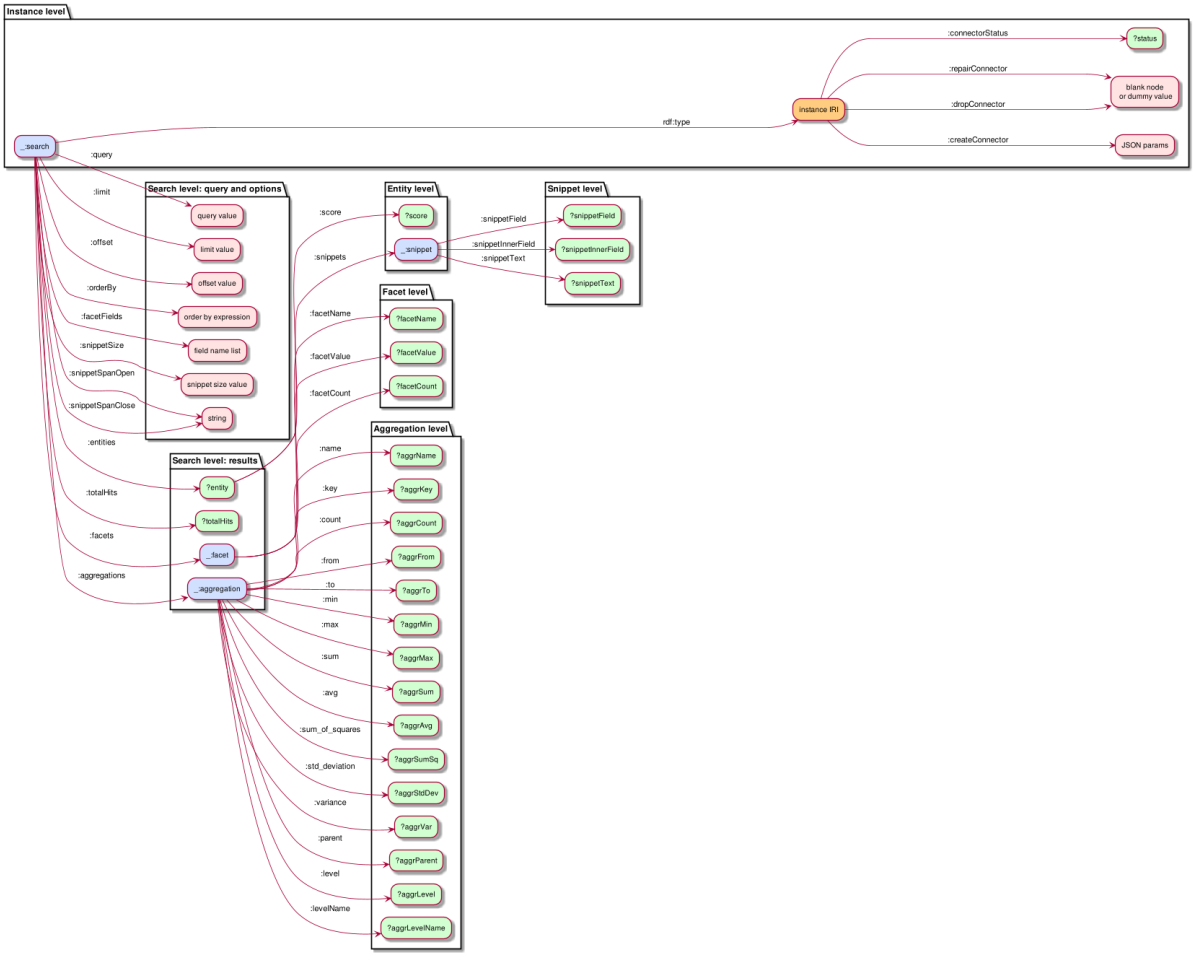
It needs to be rewritten like this:

- Per-field rule on field `location`: `$this = <urn:Europe>`
- Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
- Top-level document filter: `BOUND(?location)`

4.5.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to. For example, to retrieve entities, you need to use `:entities` on a search instance, and to retrieve snippets, you need to use `:snippets` on an entity.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.5.8 Caveats

4.5.8.1 Order of control

Even though SPARQL per se is not sensitive to the order of triple patterns, the Elasticsearch GraphDB Connector expects to receive certain predicates before others so that queries can be executed properly. In particular, predicates that specify the query or query options must come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.

4.5.9 Upgrading from previous versions

4.5.9.1 Migrating from GraphDB 9.x

GraphDB 10.0 introduces major changes to the filtering mechanism of the connectors. Existing connector instances will not be usable and attempting to use them for queries or updates will throw an error.

If your GraphDB 9.x (or older) connector definitions **do not** include an entity filter, you can simply *repair them*.

If your GraphDB 9.x (or older) connector definitions **do** include an entity filter with the `entityFilter` option, you need to rewrite the filter with one of the current *filter types*:

1. Save your existing connector definition.
2. Drop the connector instance.

3. In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter. Rewrite the filters as follows.

   Rule of thumb:

   - If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

   So if we take the example:

   ```
   ?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
   ```

   It needs to be rewritten like this:

   - Per-field rule on field `location`: `$this = <urn:Europe>`
   - Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
   - Top-level document filter: `BOUND(?location)`

4. Recreate the connector instance using the new definition.

4.6 OpenSearch GraphDB Connector

Note: This feature requires a *GraphDB Enterprise license*.

Note: GraphDB supports *full-text search* options as well.

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as OpenSearch but with the additional benefit of staying automatically up to date with the GraphDB repository data.

The Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

OpenSearch also allows for *vector search*, which uses semantic similarity to ensure better recall than full-text search. The embeddings used for this vector search are created by an embedding model client configured in the connector, with one of the options being a client for the default local model in the Graphwise Transformer tool. You can learn more about configuring such a model under *Embedding model configuration for vector search*.

The main features of the GraphDB Connectors are:

- maintenance of an index that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entity synchronization defined by:
  - a list of fields (on the OpenSearch side) and property chains (on the GraphDB side) whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization (the default is all languages)
  - additional filtering by property and value
- full-text search using native OpenSearch queries

- snippet extraction: highlighting of search terms in the search result
- faceted search
- sorting by any preconfigured field
- paging of results using `OFFSET` and `LIMIT`
- custom mapping of RDF types to OpenSearch types;

Each feature is described in detail below.

See also:

Embedding model configuration for vector search

4.6.1 Usage

All interactions with the OpenSearch GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the OpenSearch GraphDB Connector, this is `http://www.ontotext.com/connectors/opensearch#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/opensearch#createConnector`) to create a connector instance for OpenSearch.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For OpenSearch, the instance IRI is `http://www.ontotext.com/connectors/opensearch/instance#`.

> **Warning:** Deleting the repository does not remove the indexes in OpenSearch. In order to create a connector instance with the same name as one of the instances in the deleted repository, you may need to delete the corresponding OpenSearch index first.

> **Warning:** Changing the OpenSearch URL will not reindex the data automatically. If you need the data to be reindexed, you can *repair* the connector instance.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito, and Rozova, as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
```

(continues on next page)

(continued from previous page)

```
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

4.6.2 Setup and maintenance

4.6.2.1 Prerequisites

Third-party component versions

This version of the OpenSearch GraphDB Connector uses OpenSearch version 3.1.0 (Java client) and 3.1.0 (other components).

Prerequisites for vector search

A *configured embedding model*, which will create the embeddings for vector search.

4.6.2.2 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- an OpenSearch instance to synchronize to
- classes to synchronize
- properties to synchronize

The configuration data must be provided as a JSON string representation and passed together with the create command.

You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill out the configuration form.

   If you are using the *Wine example dataset*, you can provide the following configuration parameters:

   - *Name*: `my_index`
   - *Fields*:
     - *Field name*: `grape`
     - *Property chain*:
       * `http://www.example.com/demo/wine#madeFromGrape`
       * `http://www.w3.org/2000/01/rdf-schema#label`
     - *Nested object fields*: Select *Indexed*, *Stored*, *Analyzed* and *Multivalued*
   - *Types*: `http://www.example.com/demo/wine#Wine`
   - *Opensearch node*: `http://localhost:9200` (or the address of your OpenSearch instance)
   - Select *Opensearch cluster sniff*, *Opensearch manage index* and *Opensearch manage mappings*

- *Opensearch bulk update batch size*: `5000`
- *Opensearch index create settings*: `5242880`

4. If you want to use vector search with your connector, you need to configure the fields for similarity search.
   1. Provide the *embeddings model you want to use*.
   2. Specify `vector` as the datatype of the field to use for similarity search. See the *documentation for the datatype field* below for more information.
5. Execute the `CREATE` statement from the form by clicking *OK*. Alternatively, you can view its SPARQL query by clicking *View SPARQL Query*, and then copy it to execute it manually or integrate it in automation scripts.

   See also:

   List of creation parameters provides thorough explanation of all creation parameters.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `opensearch:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
    opensearch-index:my_index opensearch:createConnector '''
{
  "opensearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ]
}
''' .
}
```

The above command creates a new OpenSearch connector instance that connects to the OpenSearch instance accessible at port `9200` on the localhost as specified by the `opensearchNode` key.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to OpenSearch. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: "grape", "sugar", and "year". The field names are used later in the queries.

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine's `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to the wine.

The fields `sugar` and `year` contain discrete values such as *medium*, *dry*, *2012*, and *2013*, so it is best to specify the option `analyzed: false` as well. See `analyzed` in *Defining fields* for more information.

Mapping and index management

By default, GraphDB manages (creates, deletes, or updates if needed) the OpenSearch index and the OpenSearch mapping. This makes it easier to use OpenSearch as everything is done automatically. This behavior can be changed by the following options:

- `manageIndex`: if `true`, GraphDB manages the index. `true` by default.
- `manageMapping`: if `true`, GraphDB manages the mapping. `true` by default.

Note: If either of the options is set to `false`, you have to create, update or remove the `index/mapping` and, in case OpenSearch is misconfigured, the connector instance will not function correctly.

Using a non-managed schema

The present version provides no support for changing some advanced options, such as stop words, on a per-field basis. The recommended way to do this for now is to manage the mapping yourself and tell the connector to just sync the object values in the appropriate fields. Here is an example:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
    opensearch-index:my_index opensearch:createConnector '''
{
  "opensearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
```

(continues on next page)

(continued from previous page)

```
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ],
      "analyzed": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ],
      "analyzed": false
    }
  ],
  "manageMapping": false
}
''' .
}
```

This creates the same connector instance as above but it expects fields with the specified field names to be already present in the index mapping, as well as some internal GraphDB fields. For the example, you must have the following fields:

field name	OpenSearch config
`_graphdb_id`	“type”:”long”, “index”:true, “store”:true
`grape`	“type”:”text”, “index”:true, “store”:true
`sugar`	“type”:”keyword”, “index”:true, “store”:true
`year`	“type”:”keyword”, “index”:true, “store”:true

`_graphdb_id` is used internally by GraphDB and is always required.

Working with secured OpenSearch

GraphDB allows the access of a secured OpenSearch instance by passing the arbitrary `opensearchBasicAuthUser` and `opensearchBasicAuthPassword` parameters.

Instead of supplying the username and password as part of the connector instance configuration, you can also implement a custom authenticator class and set it via the `authenticationConfiguratorClass` option. See these connector authenticator examples for more information and example projects that implement such a custom class.

See the *List of creation parameters* for more information.

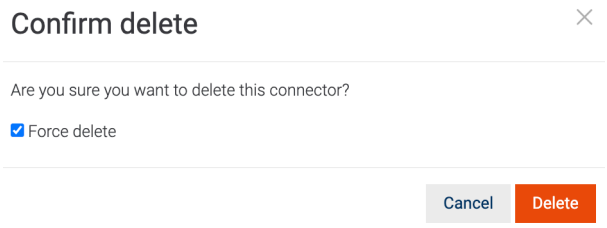
4.6.2.3 Dropping a connector instance

Dropping a connector instance removes all references to its external store from GraphDB as well as the OpenSearch index associated with it.

The drop command is triggered by a SPARQL `INSERT` with the `opensearch:dropConnector` predicate where the name of the connector instance must be in the subject position. For example, the following removes the connector `my_index`:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.6.2.4 Retrieving the create options for a connector instance

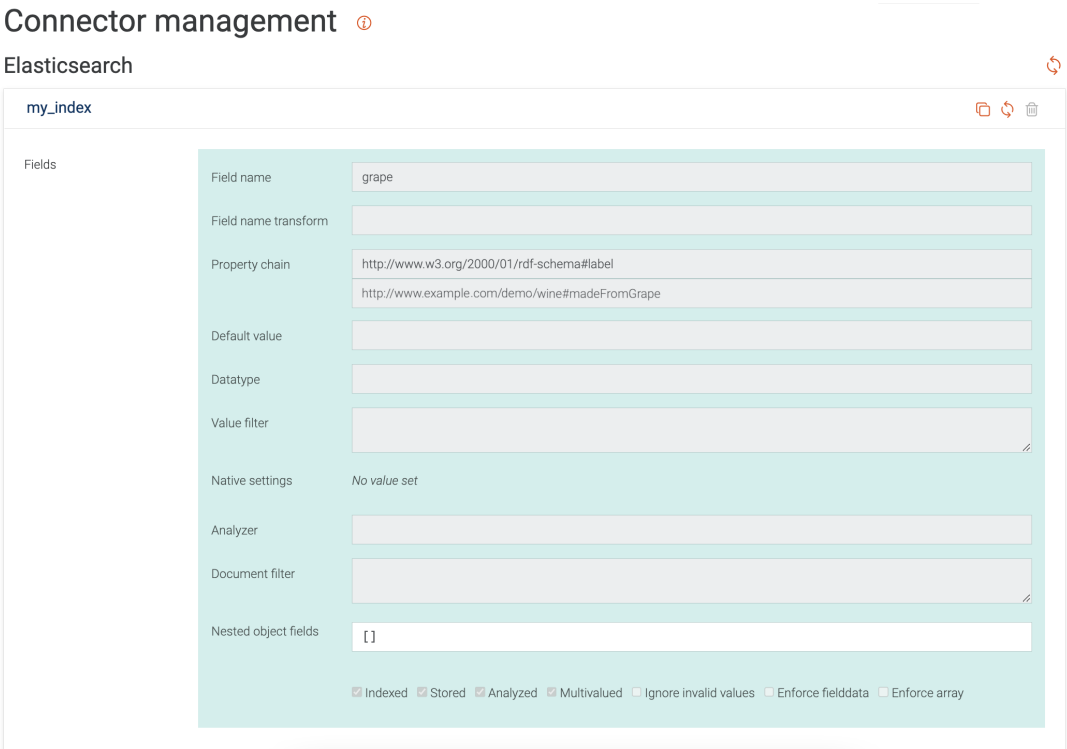
You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?createString {
  opensearch-index:my_index opensearch:listOptionValues ?createString .
}
```

4.6.2.5 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* or *Delete* icons to perform these operations. Click the *Copy* icon to copy the connector definition query to your clipboard.



With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>

SELECT ?cntUri ?cntStr {
  ?cntUri opensearch:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/opensearch/instance#my_index`, while `?cntStr` is bound to a string representing the local name such as "my_index" in this example.

4.6.2.6 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>

SELECT ?cntUri ?cntStatus {
  ?cntUri opensearch:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.6.3 Working with data

4.6.3.1 Adding, updating and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` update requests. This is achieved by intercepting all changes in the plugin and determining which OpenSearch documents need to be updated.

4.6.3.2 Simple queries

Once a connector instance has been created, you can use SPARQL to query data from it. For each matching OpenSearch document, the connector instance returns the document subject. In its simplest form, you query that data using a `SELECT` query that includes an OpenSearch query as the object of the `opensearch:query` predicate:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity {
  ?search a opensearch-index:my_index ;
      opensearch:query "grape:cabernet" ;
      opensearch:entities ?entity .
}
```

The result of this query binds `?entity` to the two wines made from grapes that have "cabernet" in their name: `:Yoyowine` and `:Franvino`.

Note: You must use the field names you chose when you created the connector instance. They can be identical to the property IRIs but you must escape any special characters according to what OpenSearch expects.

The creation of such a query follows these basic steps:

1. Get a query instance of the requested connector instance by using the RDF notation "`X a Y` (= `X rdf:type Y`), where `X` is a variable and `Y` is a connector instance IRI. `X` is bound to a query instance of the connector instance.
2. Assign a query to the query instance by using the system predicate `opensearch:query`.
3. Request the matching entities through the `opensearch:entities` predicate.

You can also provide per-query search options by using one or more option predicates. The option predicates are described in detail below.

4.6.3.3 Raw queries

To access an OpenSearch query parameter that is not exposed through a special predicate, use a raw query. Instead of providing a full-text query in the `:query` part, specify raw OpenSearch parameters. For example, to boost some parts of your full-text query as described in the OpenSearch documentation for advanced filter options, execute the following query:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity {
  ?search a opensearch-index:my_index ;
      opensearch:query '''
      {
        "query" : {
          "bool" : {
            "should" : [ {
              "query_string" : {
                "query" : "<full-text-query-not-boosted>"
              }
            }, {
              "query_string" : {
                "query" : "<full-text-query-boosted>",
                "boost" : 4.0
              }
            } ]
          }
        }
      }
      ''' ;
      opensearch:entities ?entity .
}
```

Combining OpenSearch results with GraphDB data

You can use the bound `?entity` variable in other SPARQL triple patterns to build complex queries that join to or fetch additional data from GraphDB. For example, to see the actual grapes in the matching wines as well as the year they were made:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?entity ?grape ?year {
  ?search a opensearch-index:my_index ;
      opensearch:query "grape:cabernet" ;
      opensearch:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
  ?entity wine:hasYear ?year
}
```

The result looks like this:

	entity	grape	year
1	wine:Yoyowine	wine:CabernetSauvignon	"2013"^^xsd:integer
2	wine:Franvino	wine:Merlot	"2012"^^xsd:integer
3	wine:Franvino	wine:CabernetFranc	"2012"^^xsd:integer

Note: `:Franvino` is returned twice because it is made from two different grapes, both of which are returned.

Entity match score

It is possible to access the match score returned by OpenSearch with the `score` predicate. As each entity has its own score, the predicate should come at the entity level. For example:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity ?score {
  ?search a opensearch-index:my_index ;
      opensearch:query "grape:cabernet" ;
      opensearch:entities ?entity .
  ?entity opensearch:score ?score
}
```

The result looks like this but the actual score might be different as it depends on the specific OpenSearch version:

	entity	score
1	wine:Yoyowine	"0.3979403078556061"^^xsd:double
2	wine:Franvino	"0.33036553859710693"^^xsd:double

4.6.3.4 Vector search queries

You can conduct vector searches in OpenSearch by using the similarity predicate in SPARQL, or with raw Knn (K nearest neighbors) queries.

Vector search using the similarity predicate

You can use SPARQL to query data using the `similarity` predicate. The queries are the same regardless of what creates the embeddings.

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>
SELECT DISTINCT ?entity ?score WHERE {
    ?search a opensearch-index:my-index ;
    opensearch:similarity '''myvectorfield:mysearchterm''' ;
    opensearch:entities ?entity .
    ?entity opensearch:score ?score .
}
```

Vector search using raw Knn queries

You can also use raw Knn queries to access data not exposed through a special predicate.

```
PREFIX inst: <http://www.ontotext.com/connectors/opensearch/instance#>
PREFIX conn:<http://www.ontotext.com/connectors/opensearch#>


SELECT ?totalHits WHERE {
    ?search a inst:otkg-vector ;
                    conn:query '''{
                            "query": {
                              "knn": {
                               "myvectorfield": {
                                 "vector": "mysearchterm",
                                 "k": 15,
                                 "method_parameters": {
                                    "ef_search": 150
                                 }
                               }
                             }
                            }
                          }''' .
}
```

Vector search in nested objects

If you have nested objects with `"vector"` fields, you can also execute vector queries that will search in the nested objects. To do so, you need to define the query as a nested query and provide the full path to the object in the query.

See also:

Although it focuses on Elasticsearch, the tutorial on *Vector search in nested fields with Elasticsearch* can also prove useful for users of OpenSearch. It provides an in-depth example on how to create a connector instance with vector fields in nested objects and execute queries, including multi-level nested objects.

4.6.3.5 Basic facet queries

You can also query facets with your connector instance. For example, with the sample `my_index` connector described above:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?facetName ?facetValue ?facetCount WHERE {
  # note empty query is allowed and will just match all documents, hence no opensearch:query
  ?r a opensearch-index:my_index ;
    opensearch:facetFields "year,sugar" ;
    opensearch:facets [
    opensearch:facetName ?facetName;
    opensearch:facetValue ?facetValue;
    opensearch:facetCount ?facetCount
  ]
}
```

It is important to specify the facet fields by using the `facetFields` predicate. Its value is a simple comma-delimited list of field names. In order to get the faceted results, use the `opensearch:facets` predicate. Because each facet has three components (`name`, `value`, and `count`), the `opensearch:facets` predicate returns multiple nodes that can be used to access the individual values for each component through the predicates `facetName`, `facetValue`, and `facetCount`.

The resulting bindings will look like this:

	facetName	facetValue	facetCount
1	"year"	"2012"	"3"^^xsd:long
2	"year"	"2013"	"2"^^xsd:long
3	"sugar"	"dry"	"3"^^xsd:long
4	"sugar"	"medium"	"2"^^xsd:long

You can easily see that there are three wines produced in 2012 and two in 2013. You also see that three of the wines are dry, while two are medium. However, it is not necessarily true that the three wines produced in 2012 are the same as the three dry wines as each facet is computed independently.

Tip: Faceting by analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens, and faceting uses each token to create a faceting bucket. For example, "North America" and "Europe" produce three buckets: "north", "america", and "europe", corresponding to each token in the two values. If you need to facet by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.6.3.6 Advanced facet and aggregation queries

While basic faceting allows for simple counting of documents based on the discrete values of a particular field, there are more complex faceted or aggregation searches in OpenSearch. The OpenSearch GraphDB Connector provides a mapping from OpenSearch results to RDF results but no mechanism for specifying the queries other than executing *Raw queries*.

Supported OpenSearch facets and aggregations

The OpenSearch GraphDB Connector supports mapping of the following facets and aggregations:

- Facets: terms, histogram, date histogram;
- Aggregations: terms, histogram, date histogram, range, min, max, sum, avg, stats, extended stats, value count.

For aggregations, the connector also supports sub-aggregations.

Tip: For more information on each supported facet or aggregation type, refer to the OpenSearch documentation.

RDF mapping of the results

The results are accessed through the predicate `aggregations` (much like the basic facets are accessed through `facets`). The predicate binds multiple blank nodes that each contains a single aggregation bucket. The individual bucket items can be accessed through these predicates:

Predicate	Meaning	OpenSearch counterpart
`:name`	Bucket name	`getName()`
`:key`	Key or value associated with the bucket	`getValue()` or `getKey()`
`:count`	Count of documents in the bucket	`getDocCount()`, `getValue()`
`:from`	Start of range	`getFrom()`, `getFromAsDate()`
`:to`	End of range (RangeFacet)	`getTo()`, `getToAsDate()`
`:min`	Minimum value	`getMin()`, `getValue()`
`:max`	Maximum value	`getMax()`, `getValue()`
`:sum`	Sum value	`getSum()`, `getValue()`
`:avg`	Average value	`getAvg()`, `getValue()`
`:sum_of_squares`	Sum of squares value	`getSumOfSquares()`
`:variance`	Variance value	`getVariance()`
`:std_deviation`	Standard deviation value	`getStdDeviation()`
`:parent`	Sub-aggregations: points to the parent (upper level) blank node	
`:level`	Sub-aggregations: level number where 1 is the uppermost level and the following levels are 2, 3 and so on	
`:levelName`	Sub-aggregations: level name	`getKey()` or `getValue()`

4.6.3.7 Sorting

You can sort the entities returned by a connector query according to one or more fields. To do this, assign a comma-delimited list of sort fields to the `orderBy` predicate. Each field can be prefixed with a minus to indicate sorting in descending order. For example:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>
SELECT ?entity ?sugar{
  ?search a opensearch-index:my_index ;
      opensearch:query "year:2013" ;
      opensearch:orderBy "-sugar" ;
      opensearch:entities ?entity.
  ?entity wine:hasSugar ?sugar
}
```

The result of this sample query contains wines produced in 2013 sorted according to their sugar content in descending order:

	entity	sugar
1	wine:Rozova	"medium"
2	wine:Yoyowine	"dry"

By default, entities are sorted according to their matching score in descending order.

Note: If you join the entity from the connector query to other triples stored in GraphDB, GraphDB might scramble the order. Avoid this by using SPARQL's `ORDER BY` instruction.

Tip: Sorting by an analyzed textual field works but might produce unexpected results. Analyzed textual fields are composed of tokens and sorting uses the least (in the lexicographical sense) token. For example, "North America" will be sorted before "Europe" because the token "america" is lexicographically smaller than the token "europe". If you need to sort by a textual field and still do full-text search on it, it is best to create a copy of the field with the setting `"analyzed": false`. For more information, see *Copy fields*.

4.6.3.8 Limit and offset

Limit and offset are supported on the OpenSearch side of the query. You can set these values using the predicates `limit` and `offset`. Consider this example in which an offset of `1` and a limit of `1` are specified:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity {
  ?search a opensearch-index:my_index ;
      opensearch:query "sugar:dry" ;
      opensearch:offset "1" ;
      opensearch:limit "1" ;
      opensearch:entities ?entity .
}
```

`offset` is counted from 0. The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:

	entity
1	wine:Yoyowine
2	wine:Franvino
3	wine:Blanquito

Note: The specific order in which GraphDB returns the results depends on how OpenSearch returns the matches, unless sorting is specified.

4.6.3.9 Snippet extraction

Snippet extraction lets you retrieve highlighted snippets of text that match a query. You can access the snippets by using the the dedicated predicate `opensearch:snippets`. This binds a blank node that in turn provides the actual snippets via the predicates `opensearch:snippetField` and `opensearch:snippetText`. The predicate snippets must be attached to the entity, as each entity has a different set of snippets. For example, in a search for Cabernet:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a opensearch-index:my_index ;
      opensearch:query "grape:cabernet" ;
      opensearch:entities ?entity .
  ?entity opensearch:snippets ?snippet .
  ?snippet opensearch:snippetField ?snippetField ;
     opensearch:snippetText ?snippetText .
}
```

the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

	entity	snippetField	snippetText
1	wine:Yoyowine	"grape"	"Cabernet Sauvignon"
2	wine:Franvino	"grape"	"MerlotCabernet Franc"

Note: The actual snippets might be different depending on the specific OpenSearch implementation.

You can tweak how the snippets are collected and composed by using the following option predicates:

- `opensearch:snippetSize`: sets the maximum size of the extracted text fragment, `250` by default
- `opensearch:snippetSpanOpen`: the text to insert before the highlighted text, `<em>` by default
- `opensearch:snippetSpanClose`: the text to insert after the highlighted text, `</em>` by default

The option predicates are set on the query instance, much like the `opensearch:query` predicate.

Snippets from nested documents

Note: This section uses the *Nested objects* dataset described further down below in this page.

Snippets extracted from nested documents (when a nested query is used) will be available through the same mechanism as snippets from non-nested fields. In addition, nested snippet results provide the nested search path via the `snippetInnerField` predicate. For example, in a nested search on the field "grandChildren" (specified by "path") and a match query for "tylor" on the nested field "grandChildren.name":

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity ?snippetInnerField ?snippetField ?snippetText {
  ?search a opensearch-index:my_index ;
      opensearch:query '''
        {
           "query":{
              "nested":{
                 "path":"grandChildren",
                 "query":{
                    "bool":{
                       "must":[
                          {
                             "match":{
                                "grandChildren.name":"tylor"
                             }
                          }
                       ]
                    }
                 }
              }
           }
        }
      ''' ;
      opensearch:entities ?entity .
  ?entity opensearch:snippets ?snippet .
  ?snippet opensearch:snippetInnerField ?snippetInnerField ;
           opensearch:snippetField ?snippetField ;
           opensearch:snippetText ?snippetText .
}
```

the query returns all people who have a grandchild whose name matches "tylor", as well as the highlighted snippets:

?entity	?snippetInnerField	?snippetField	?snippetText
urn:Eva	grandChildren	grandChildren.name	John-Tylor
urn:John	grandChildren	grandChildren.name	John-Tylor
urn:John	grandChildren	grandChildren.name	Tylor
urn:Mary	grandChildren	grandChildren.name	Tylor

Note that the matching field whose matching values are highlighted is provided via the `snippetField` predicate, just like extracting snippets with non-nested searches, while the predicate `snippetInnerField` provides the field on which the nested search was executed.

4.6.3.10 Total hits

You can get the total number of matching OpenSearch documents (hits) by using the `opensearch:totalHits` predicate. For the `my_index` connector instance used above, a query for all wines made in 2012 might look like this:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?totalHits {
    ?r a opensearch-index:my_index ;
       opensearch:query "year:2012" ;
       opensearch:totalHits ?totalHits .
}
```

As there are three wines made in 2012, the value `3` (of type `xsd:long`) binds to `?totalHits`.

As you see above, you can omit the return of any of the matching entities. This can be useful if there are many hits and you want to calculate pagination parameters.

4.6.4 List of creation parameters

The creation parameters define how the `opensearch:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object, where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the *Create Connector user interface* without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal Lucene data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to make an OpenSearch index from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `opensearch:graph`, where the prefix `opensearch:` represents the IRI `http://www.ontotext.com/connectors/opensearch#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *read-only mode*.

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
INSERT {
    GRAPH opensearch:graph {
        ...
    }
} WHERE {
        ...
};
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>
INSERT DATA {
```

(continues on next page)

(continued from previous page)

```
    opensearch-index:my_index opensearch:createConnector '''
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponding to a single predicate with a field name the same as the predicate, so you must use escaping when issuing OpenSearch queries.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`opensearchNode` (string), required, the OpenSearch instance to sync to As OpenSearch is a third-party service, you have to specify the node where it is running. The node value takes the form `http://hostname.domain:port` or `https://hostname.domain:port`. No default value. Can be updated at runtime with no need to rebuild the index.

> **Note:** OpenSearch exposes two protocols — the native transport* protocol over port `9300` and the RESTful API over port `9200`. The OpenSearch GraphDB Connector uses the RESTful API over port `9200`.

`indexCreateSettings` (json), optional, the settings for creating the OpenSearch index This option is passed directly to OpenSearch when creating the index.

`opensearchBasicAuthUser` (string), optional, the settings for supplying the authentication user No default value. Can be updated at runtime with no need to rebuild the index.

`opensearchBasicAuthPassword` (string), optional, the settings for supplying the authentication password A password is a string with a single value that is not logged or printed. No default value. Can be updated at runtime with no need to rebuild the index.

`opensearchClusterSniff` (boolean), controls whether to build the server address list by sniffing on the OpenSearch cluster Corresponds to the OpenSearch `client.transport.sniff` option. `true` by default. Can be updated at runtime with no need to rebuild the index.

`bulkUpdateBatchSize` (integer), controls the maximum number of documents sent per bulk request Default value is `5,000`. Can be updated at runtime with no need to rebuild the index.

`authenticationConfiguratorClass` provides custom authentication behavior Optional.

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type. See also the examples in *General full-text search with the connectors*.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual, but only some of the languages represented in the literal values can be mapped. You can do this by specifying a list of

language ranges to be matched to the language tags of literals according to RFC 4647, Section 3.3.1. Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to OpenSearch The fields specify exactly which parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in OpenSearch. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

- **`fieldName` (string), required, the name of the field in OpenSearch** The name of the field defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field name is used at query time to refer to the field. There are few restrictions on the allowed characters in a field name, but to avoid unnecessary escaping (which depends on how OpenSearch parses its queries), we recommend keeping the field names simple.
- **`fieldNameTransform` (one of `none`, `predicate` or `predicate.localName`), optional, `none` by default** Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

  See *Indexing all literals in distinct fields* for an example.
- **`propertyChain` (list of IRI), required, defines the property chain to reach the value** The property chain defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple, etc. In this model, a property chain with a single element corresponds to a direct property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

  The IRI of the document will be synchronized to the special field `_id` in OpenSearch. You may use it to query OpenSearch directly and to retrieve the matching entity IRI.

  See *Copy fields* for defining multiple fields with the same property chain.

  See *Multiple property chains per field* for defining a field whose values are populated from more than one property chain.

  See *Indexing language tags* for defining a field whose values are populated with the language tags of literals.

  See *Indexing the IRI of an entity* for defining a field whose values are populated with the IRI of the indexed entity.

  See *Wildcard literal indexing* for defining a field whose values are populated with literals regardless of their predicate.
- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.
- **`documentFilter` (string), optional, specifies the nested document filter for the field** Only for fields that define nested documents. See also *Entity filtering*.
- **`defaultValue` (string), optional, specifies a default value for the field** Provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or a IRI. It has no default value.
- **`indexed` (boolean), optional, default `true`** If indexed, a field is available for OpenSearch queries. `true` by default.

If `true`, this option corresponds to `"index" = true`. If `false`, it corresponds to `"index" = false`.

- **`stored` (boolean), optional, default `true`** Whether fields are stored in OpenSearch. `true` by default.

  This option corresponds to the property `"store"` in the OpenSearch mapping.

- **`analyzed` (boolean), optional, default `true`** When literal fields are indexed in OpenSearch, they will be analyzed according to the analyzer settings. Should you require that a given field is not analyzed, you may use `"analyzed"`. This option has no effect for IRIs, which are never analyzed. `true` by default.

  If `true`, this option will use automatic or manual (`datatype` option) type for the OpenSearch mapping. If `false`, it corresponds to `"type" = "keyword"` (i.e., the default type will be changed to keyword).

- **`multivalued` (boolean), optional, default `true`** RDF properties and synchronized fields may have more than one value. If `multivalued` is set to `true`, all values will be synchronized to OpenSearch. If set to `false`, only a single value will be synchronized. `true` by default.

- **`ignoreInvalidValues` (boolean), optional, default `false`** Per-field option that controls what happens when a value cannot be converted to the requested (or previously detected) type. `False` by default.

  Example use: when an invalid date literal like `"2021-02-29"^^xsd:date` (2021 is not a leap year) needs to be indexed as a date, or when an IRI needs to be indexed as a number.

  > **Note:** Some conversions are always valid: any literal to an FTS field, any non-literal (IRI, blank node, embedded triple) to a non-analyzed field. When `true`, such values will be skipped with a note in the logs. When `false`, such values will break the transaction.

- **`array` (boolean), optional, default `false`** Normally, OpenSearch creates an array only if more than value is present for a given field. If `array` is set to `true`, OpenSearch will always create an array even for single values. If set to `false`, OpenSearch will create arrays for multiple values only. `False` by default.

- **`fielddata` (boolean), optional, default `false`** Allows fielddata to be built in memory for text fields. This can consume a lot of heap space, especially when loading high cardinality text fields. `False` by default.

- **`datatype` (string), optional, the manual datatype override** By default, the OpenSearch GraphDB Connector uses datatype of literal values to determine how they should be mapped to OpenSearch types. For more information on the supported datatypes, see *Datatype mapping*.

  You can override the datatype mapping with the `"datatype"` parameter, which can be specified per field. The value of `datatype` can be any of the `xsd:` types supported by the automatic mapping or a native OpenSearch type prefixed by `native:`. For example, both `xsd:long` and `native:long` map to the long type in OpenSearch.

  When using vector similarity search, you need to provide the datatype of the field that will be used for similarity search by providing `"vector"` as the value of the `"datatype"` parameter. This datatype creates a `knn_vector` field.

  You can also create a single vector from multiple fields by specifying `"vector"` as the datatype for each of the fields you want to use.

- **`nativeSettings` (json), optional, custom field settings** The setting for the OpenSearch mappings and field types of the respective field such as the `format` of the datatype. Native field settings require an explicit native datatype.

  `nativeSettings` are not allowed for the following parameters so as to avoid conflicts with the existing way to specify them: `type`, `index`, `store`, `analyzer`, `fielddata`.

- **`objectFields` (objects array), optional, nested object mapping** When `native:object`, `native:nested`, or `native:geo_point` is used as a datatype value, this provides a mapping for the nested object's fields. If `datatype` is not provided, then `native:object` will be assumed.

For the difference between `object` and `nested`, refer to the OpenSearch nested field type. The `geo_point` type must have exactly two fields named `lat` and `long` (required by OpenSearch, see geo-point field type).

Nested objects support further nested objects with a limit of five levels of nesting. See *Nested objects* for an example.

- **`startFromParent` (integer), optional, default `0`** Start processing the property chain from the N-th parent instead of the root of the current nested object. 0 is the root of the current nested object, 1 is the parent of the nested object, 2 is the parent of the parent and so on.
- **`analyzer` (string), optional, per field analyzer** The OpenSearch analyzer that is used for indexing the field can be specified with the parameter `analyzer`. It will be passed directly to OpenSearch's property `analyzer` when creating the mapping (see Text Analyzers in the OpenSearch documentation). For example:

```
{
...
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzer": "my_analyzer"
    },
...
}
```

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

`embeddingModel` (string), optional, specifies the embedding model used to create the embeddings when using vector search See also *Embedding model configuration for vector search*.

4.6.4.1 Updating parameters at runtime

As mentioned above, you can update the following connection parameters at runtime with no need to rebuild the index:

- `opensearchNode`
- `opensearchClusterSniff`
- `opensearchBasicAuthUser`
- `opensearchBasicAuthPassword`
- `bulkUpdateBatchSize`

The following SPARQL update request demonstrates this by updating the user and password values:

```
PREFIX conn:<http://www.ontotext.com/connectors/opensearch#>
PREFIX inst:<http://www.ontotext.com/connectors/opensearch/instance#>
INSERT DATA {
inst:proper_index conn:updateConnector '''
    {
        "opensearchBasicAuthUser": "foo",
        "opensearchBasicAuthPassword": "bar"
    }
''' .
}
```

4.6.4.2 Special field definitions

Nested objects

Nested objects are OpenSearch documents that are used as values in the main document or in other nested objects. Up to five levels of nesting is possible. They are defined with the *objectFields* option.

Let's look at an example that uses the following data, which consists of children and grandchildren relations:

```
<urn:John>
    a <urn:Person> ;
    <urn:name> "John" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 60 ;
    <urn:hasSpouse> <urn:Mary> ;
    <urn:hasChild> <urn:Billy> ;
    <urn:hasChild> <urn:Annie> .

<urn:Mary>
    a <urn:Person> ;
    <urn:name> "Mary" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 58 ;
    <urn:hasSpouse> <urn:John> ;
    <urn:hasChild> <urn:Billy> .

<urn:Eva>
    a <urn:Person> ;
    <urn:name> "Eva" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 45 ;
    <urn:hasChild> <urn:Annie> .

<urn:Billy>
    a <urn:Person> ;
    <urn:name> "Billy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 35 ;
    <urn:hasChild> <urn:Tylor> ;
    <urn:hasChild> <urn:Melody> .

<urn:Annie>
    a <urn:Person> ;
    <urn:name> "Annie" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 28 ;
    <urn:hasChild> <urn:Sammy> .

<urn:Tylor>
    a <urn:Person> ;
    <urn:name> "Tylor" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 5 .

<urn:Melody>
    a <urn:Person> ;
    <urn:name> "Melody" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 2 .

<urn:Sammy>
    a <urn:Person> ;
```

(continues on next page)

(continued from previous page)

```
    <urn:name> "Sammy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 10 .

<urn:Male> <urn:label> "male" .

<urn:Female> <urn:label> "female" .
```

We can create a nested objects index that consists of children and grandchildren with their corresponding fields defining their gender and age. We use the `native:nested` type because we want to query the nested objects independently of each other:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine#gadget"],
      "fields": [
        {
          "fieldName": "name",
          "propertyChain": [
            "urn:name"
          ]
        },
        {
          "fieldName": "age",
          "propertyChain": [
            "urn:age"
          ],
          "datatype": "xsd:long"
        },
        {
          "fieldName": "hasSpouse",
          "propertyChain": [
            "urn:hasSpouse"
          ]
        },
        {
          "fieldName": "gender",
          "propertyChain": [
            "urn:gender",
            "urn:label"
          ]
        },
        {
          "fieldName": "children",
          "propertyChain": [
            "urn:hasChild"
          ],
          "datatype": "native:nested",
          "objectFields": [
            {
              "fieldName": "id",
              "propertyChain": [
                "$self"
              ]
            },
            {
```

(continues on next page)

(continued from previous page)

```
            "fieldName": "name",
            "propertyChain": [
              "urn:name"
            ]
          },
          {
            "fieldName": "age",
            "propertyChain": [
              "urn:age"
            ],
            "datatype": "xsd:long"
          },
          {
            "fieldName": "gender",
            "propertyChain": [
              "urn:gender",
              "urn:label"
            ]
          },
          {
            "fieldName": "children",
            "propertyChain": [
              "urn:hasChild"
            ],
            "objectFields": [
              {
                "fieldName": "id",
                "propertyChain": [
                  "$self"
                ]
              },
              {
                "fieldName": "name",
                "propertyChain": [
                  "urn:name"
                ]
              },
              {
                "fieldName": "age",
                "propertyChain": [
                  "urn:age"
                ],
                "datatype": "xsd:long"
              }
            ]
          }
        ]
      },
      {
        "fieldName": "grandChildren",
        "valueFilter": "$this -> type in (<urn:Person>)",
        "propertyChain": [
          "urn:hasChild",
          "urn:hasChild"
        ],
        "datatype": "native:nested",
        "objectFields": [
          {
            "fieldName": "id",
            "propertyChain": [
              "$self"
```

(continues on next page)

(continued from previous page)

```
              ]
            },
            {
              "fieldName": "name",
              "propertyChain": [
                "urn:name"
              ]
            },
            {
              "fieldName": "age",
              "propertyChain": [
                "urn:age"
              ],
              "datatype": "xsd:long"
            },
            {
              "fieldName": "gender",
              "propertyChain": [
                "urn:gender",
                "urn:label"
              ]
            }
          ]
        },
      ],
      "types": [
        "urn:Person"
      ],
      "opensearchNode": "http://localhost:9200"
    }
 '''
}
```

To find male grandchildren of age 5 and older, we will use the following query:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?entity {
    ?search a opensearch-index:my_index ;
        opensearch:query '''
            {
               "query" : {
                  "nested" : {
                     "path" : "grandChildren",
                     "query" : {
                        "bool" : {
                           "must" : [
                              {
                                 "match" : {
                                    "grandChildren.gender" : "male"
                                 }
                              },
                              {
                                 "range" : {
                                    "grandChildren.age" : {
                                       "gt" : 5
                                    }
                                 }
                              }
                           ]
```

(continues on next page)

(continued from previous page)

```
                        }
                    }
                }
            }
        }
    ''' ;
    opensearch:entities ?entity .
}
ORDER BY ?entity
```

The result looks like this:

?entity
urn:Eva
urn:John

Nested objects with vector datatypes

You can also execute vector search queries with nested objects by adding fields to the nested object with `"vector"` as the value of the `"datatype"` parameter.

See also:

- *The documentation for the datatype field below* for more information
- *Vector search in nested objects*
- The tutorial on *Vector search in nested fields with Elasticsearch*, while focusing on Elasticsearch, provides an in-depth example on creating a connector instance with vector fields in nested objects

Copy fields

It is often convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases such as faceting or sorting vs full-text search. The OpenSearch GraphDB Connector has explicit support for fields that copy their value from another field. To do this, specify a single element in the property chain of the form `@otherFieldName`, where `otherFieldName` is another non-copy field. Take the following example:

```
...
  "fields": [
    {
      "fieldName": "grape",
      "facet": false,
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "analyzed": true
    },
    {
      "fieldName": "grapeFacet",
      "propertyChain": [
        "@grape"
      ],
      "analyzed": false
    }
  ]
...
```

The snippet creates an analyzed field "grape" and a non-analyzed field "grapeFacet"; both fields are populated with the same values and "grapeFacet" is defined as a copy field that refers to the field "facet".

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in OpenSearch) with more than one property chain — for example, the concept of "name" could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in OpenSearch, you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.example.com/demo/wine#canonicalName"
      ],
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.example.com/demo/wine#historicalName"
      ]
      ...
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in OpenSearch.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old`, you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo/wine#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.example.com/demo/wine#name",
             "lang()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.example.com/demo/wine#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo/wine#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.example.com/demo/wine#name",
             "graph()"
```

(continues on next page)

(continued from previous page)

```
          ]
        }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.example.com/demo/wine#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo/wine#name"
           ]
         },
         {
           "fieldName": "feature",
           "propertyChain": [
             "http://www.example.com/demo/wine#feature",
             "localName()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.example.com/demo/wine#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
    opensearch-index:my_index opensearch:createConnector '''
{
  "opensearchNode": "localhost:9200",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "entityId",
      "propertyChain": [
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

Note: Note that GraphDB will also use the IRI of each entity as the ID of each document in OpenSearch, which is represented by the field `id`.

4.6.5 Datatype mapping

The OpenSearch GraphDB Connector maps different types of RDF values to different types of OpenSearch values according to the basic type of the RDF value (IRI or literal) and the datatype of literals. The auto-detection uses the following mapping:

RDF value	RDF datatype	OpenSearch type
IRI	n/a	`keyword`
literal	any type not explicitly mentioned below	`text`
literal	`xsd:boolean`	`boolean`
literal	`xsd:double`	`double`
literal	`xsd:float`	`float`
literal	`xsd:long`	`long`
literal	`xsd:int`	`integer`
literal	`xsd:dateTime`	`date` with `format: strict_date_time`
literal	`xsd:date`	`date` with `format: strict_date`
literal	`xsd:time`	`date` with `format: strict_time_no_millis||strict_time`
literal	`xsd:gYear`	`date` with `format: strict_year`
literal	`xsd:gYearMonth`	`date` with `format: strict_year_month`

For any given field, the automatic mapping uses the first value it sees. This works fine for clean datasets but might lead to problems if your dataset has non-normalized data — for example, the first value has no datatype but other values do. Because of this, it's best to set the *datatype* to a fixed value such as `xsd:date`.

Note: There is one additional datatype — `"vector"` — used for vector search but not by the datatype mapping mechanism. See the *datatype entry in the Creation parameters section* for more details.

Warning: The commonly used `xsd:integer` and `xsd:decimal` datatypes are not indexed as numbers because they represent infinite precision numbers. You can override that by using the *datatype* option to cast them to `xsd:long`, `xsd:double`, or `xsd:float` as appropriate.

4.6.5.1 Date and time conversion

RDF and OpenSearch use slightly different models for representing dates and times, even though the values might look very similar.

Years in RDF values use the XSD format and are *era* years, where positive values denote the common era and negative values denote years before the common era. There is no year zero.

Years in OpenSearch use the ISO 8601 format and are *proleptic* years: positive values denote years from the common era with any previous eras just going down by one mathematically so there is year zero.

In short:

- year 2020 CE = year 2020 in XSD = year 2020 in ISO.
- …
- year 1 CE = year 1 in XSD = year 1 in ISO.
- year 1 BCE = year -1 in XSD = year 0 in ISO.
- year 2 BCE = year -2 in XSD = year -1 in ISO.
- …

All years coming from RDF literals will be converted to ISO before indexing in OpenSearch.

Both XSD and ISO date and time values support timezones. In addition to that, XSD defines the lack of a timezone as *undetermined*. Since we do not want to have any undetermined state in the indexing system, GraphDB defines the undetermined time zone as UTC. For example, `"2020-02-14T12:00:00"^^xsd:dateTime` is equivalent to `"2020-02-14T12:00:00Z"^^xsd:dateTime` (Z is the UTC timezone, also known as +00:00).

Also note that XSD dates and partial dates, e.g., `xsd:gYear` values, may have a timezone, which leads to additional complications. For example, `"2020+02:00"^^xsd:gYear` (the year 2020 in the +02:00 timezone) will be normalized to `2019-12-31T22:00:00Z` (the previous year!) if GraphDB followed strict timezone adherence. We have chosen to ignore the timezone on any values that do not have an associated time value. For example, all of the following values will be treated as if they specified UTC as their timezone:

- `"2020-02-15+02:00"^^xsd:date`
- `"2020-02+02:00"^^xsd:gYearMonth`
- `"2020+02:00"^^xsd:gYear`

4.6.6 Entity filtering

The OpenSearch connector supports four kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to OpenSearch if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable.

4.6.6.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

On nested documents, the per-field value filter can be used to remove the entire nested document early in the indexing process, e.g., by checking the type of the nested document via next hop with `rdf:type`.

Nested document filter The nested document filter is specified with the `documentFilter` parameter **inside the field definition** of the field that defines the root of a nested document. The filter is evaluated after the entire nested document has been collected. Failing to pass this filter removes the entire nested document.

Inside a nested document filter, the field names are within the context of the nested document and not within the context of the top-level document. For example, if you have a `children` field that defines a nested document and you use a filter like `?age < "10"^^xsd:int`, it will refer to the field `children.age`. You can use the prefix `$outer.` one or more times to refer to field values from the outer document from the viewpoint of the nested document. For example, `$outer.age > "25"^^xsd:int` will refer to the `age` field that is a sibling of the `children` field.

Other than the above differences, the nested document filter is equivalent to the top-level document filter from the viewpoint of the nested document.

See also *Migrating from GraphDB 9.x*.

4.6.6.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on original RDF values before they are converted to OpenSearch values using *datatype mapping*.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

Continued on next page

Table 13 – continued from previous page

Operator	Meaning

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible such as `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height`'s value is `"150"^^xsd:int` and `dateOfBirth`'s value is `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`

Continued on next page

Table 13 – continued from previous page

Operator	Meaning
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.6.6.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.6.6.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter and the nested document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.6.6.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.example.com/demo/wine#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine#gadget"],
      "fields": [
         {
```

(continues on next page)

(continued from previous page)

```
          "fieldName": "name",
          "propertyChain": ["http://www.example.com/demo/wine#name"]
        },
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo/wine#city"],
          "valueFilter": "$this = \\"London\\""
        }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo/wine#city"],
          "defaultValue": "London"
        }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.6.6.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.example.com/demo/wine2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
    rdf:type example2:Person ;
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
    example2:taggedWith example2:Berlin ;
```

(continues on next page)

(continued from previous page)

```
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to OpenSearch, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

INSERT DATA {
  opensearch-index:my_index opensearch:createConnector '''
    {
      "opensearchNode": "localhost:9200",
      "types": ["http://www.example.com/demo/wine2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
         {
           "fieldName": "taggedWithPerson",
           "propertyChain": ["http://www.example.com/demo/wine2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo/wine2#Person>"
         },
         {
           "fieldName": "taggedWithLocation",
           "propertyChain": ["http://www.example.com/demo/wine2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo/wine2#Location>"
         }
      ]
    }
  ''' .
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

You can check this by issuing a faceted search for `taggedWithLocation` and `taggedWithPerson`:

```
PREFIX opensearch: <http://www.ontotext.com/connectors/opensearch#>
PREFIX opensearch-index: <http://www.ontotext.com/connectors/opensearch/instance#>

SELECT ?facetName ?facetValue ?facetCount {
  ?search a opensearch-index:my_index ;
      opensearch:facetFields "taggedWithLocation,taggedWithPerson" ;
      opensearch:facets [
        opensearch:facetName ?facetName ;
        opensearch:facetValue ?facetValue ;
        opensearch:facetCount ?facetCount
      ]
}
```

If the filter was applied, you should get only `:Berlin` for `taggedWithLocation` and only `:Einstein` and `:Mozart` for `taggedWithPerson`:

facetName	facetValue	facetCount
taggedWithLocation	`http://www.example.com/demo/wine2#Berlin`	3
taggedWithPerson	`http://www.example.com/demo/wine2#Mozart`	2
taggedWithPerson	`http://www.example.com/demo/wine2#Einstein`	1

4.6.6.7 Migrating filters from GraphDB 9.x

If you used entity filters in the connectors in GraphDB 9.x (or older) with the `entityFilter` option, you need to rewrite them using one of the current *filter types*.

In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter.

Rule of thumb:

- If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
- If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

So if we take the example:

```
?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
```

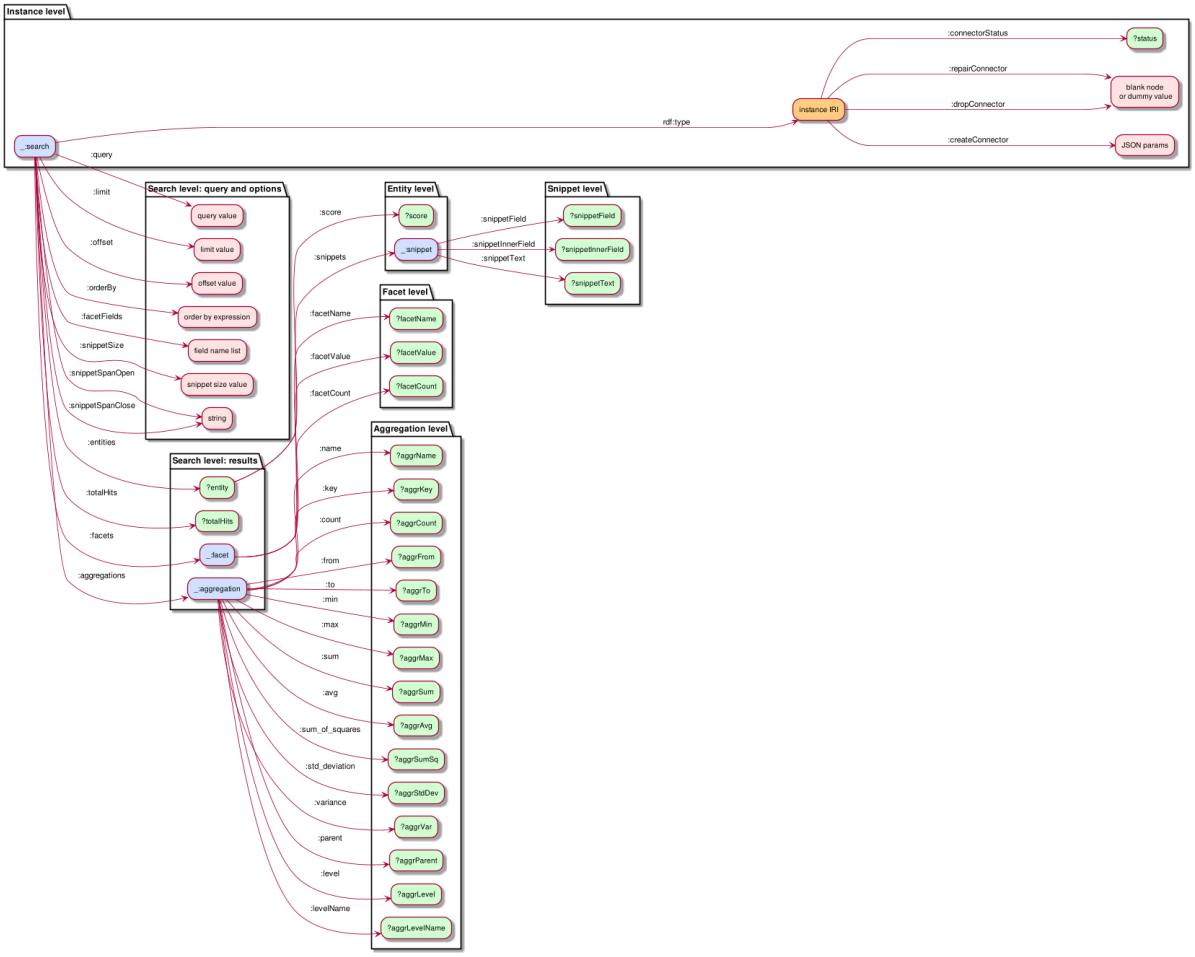
It needs to be rewritten like this:

- Per-field rule on field `location`: `$this = <urn:Europe>`
- Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
- Top-level document filter: `BOUND(?location)`

4.6.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to. For example, to retrieve entities, you need to use `:entities` on a search instance, and to retrieve snippets, you need to use `:snippets` on an entity.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.6.8 Caveats

4.6.8.1 Order of control

Even though SPARQL per se is not sensitive to the order of triple patterns, the OpenSearch GraphDB Connector expects to receive certain predicates before others so that queries can be executed properly. In particular, predicates that specify the query or query options must come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.

4.6.9 Upgrading from previous versions

4.6.9.1 Migrating from GraphDB 9.x

GraphDB 10.0 introduces major changes to the filtering mechanism of the connectors. Existing connector instances will not be usable and attempting to use them for queries or updates will throw an error.

If your GraphDB 9.x (or older) connector definitions **do not** include an entity filter, you can simply *repair them*.

If your GraphDB 9.x (or older) connector definitions **do** include an entity filter with the `entityFilter` option, you need to rewrite the filter with one of the current *filter types*:

1. Save your existing connector definition.
2. Drop the connector instance.

3. In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter. Rewrite the filters as follows.

   Rule of thumb:

   - If you want to remove individual values, i.e., if the operand **is not** `BOUND()` –> rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand **is** `BOUND()` –> rewrite with top-level document filter.

   So if we take the example:

   ```
   ?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
   ```

   It needs to be rewritten like this:

   - Per-field rule on field `location`: `$this = <urn:Europe>`
   - Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
   - Top-level document filter: `BOUND(?location)`
4. Recreate the connector instance using the new definition.

4.6.10 Migrating from Elasticsearch to OpenSearch Connector

1. Save your existing connector definition.
2. Drop the Elasticsearch connector instance.
3. Change all references from "elastic" to "open" in your connector definition query and recreate the connector.

Warning: Dropping the existing connector instance is best done before upgrading to GraphDB 10.3. Dropping instances after upgrading may result in a failure.

Note: If dropping the connector fails, you can force-drop the connector and manually delete the index in OpenSearch.

4.7 Kafka GraphDB Connector

Note: This feature requires a *GraphDB Enterprise license*.

Note: GraphDB supports *full-text search* options as well.

The Kafka connector provides a means to synchronize RDF model changes to any Kafka consumer, staying automatically up-to-date with the GraphDB repository data.

GraphDB's Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

On the Kafka side, the RDF entities are translated to JSON documents.

The main features of the Kafka Connector are:

- maintenance of a Kafka topic that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entity synchronization defined by:
  - a list of fields on the Kafka side and property chains on the GraphDB side whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization with all languages being the default
  - additional filtering by property and value

Unlike the Elasticsearch, OpenSearch, Solr, and Lucene connectors, the Kafka Connector does not have a query interface since Kafka is a simple message queue and does not provide a search feature.

In terms of Kafka terminology and behavior:

- Each connector instance must be assigned to a fixed *Kafka topic*.
- The connector is a *Kafka producer*, and does not have any information about the Kafka consumers.
- The partitions are assigned by the Kafka framework and not the connector.

4.7.1 Usage

All interactions with the Kafka GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the Kafka GraphDB Connector, this is `http://www.ontotext.com/connectors/kafka#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/kafka#createConnector`) to create a connector instance for Kafka.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For Kafka, the instance IRI is `http://www.ontotext.com/connectors/kafka/instance#`.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito, and Rozova, as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
```

(continues on next page)

(continued from previous page)

```
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

4.7.2 Setup and maintenance

4.7.2.1 Prerequisites

Third-party component versions This version of the Kafka GraphDB Connector uses Kafka version 3.9.1.

4.7.2.2 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- a Kafka node and topic to synchronize to
- classes to synchronize
- properties to synchronize

The configuration data must be provided as a JSON string representation and passed together with the create command.

You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill out the configuration form.
4. Execute the `CREATE` statement from the form by clicking *OK*. Alternatively, you can view its SPARQL query by clicking *View SPARQL Query*, and then copy it to execute it manually or integrate it in automation scripts.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `kafka:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
    kafka-inst:my_index kafka:createConnector '''
{
  "kafkaNode": "localhost:9092",
  "kafkaTopic": "my_index",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
```

(continues on next page)

(continued from previous page)

```
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
      ]
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ]
    }
  ]
}
''' .
}
```

The above command creates a new Kafka connector instance that connects to the Kafka instance accessible at port `9200` on the localhost as specified by the `kafkaNode` key.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to Kafka. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: “grape”, “sugar”, and “year”. The field names are used later in the queries.

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine’s `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to the wine.

Working with a secured Kafka broker

GraphDB can connect to a secured Kafka broker using the SASL/PLAIN authentication mechanism. To configure it, set the *kafkaPlainAuthUsername* and *kafkaPlainAuthPassword* parameters. Since the password will be transmitted in clear text, it’s best to enable SSL on the Kafka broker, and accordingly set the *kafkaSSL* parameter to `true`.

Instead of supplying the username and password as part of the connector instance configuration, you can also implement a custom authenticator class and set it via the `authenticationConfiguratorClass` option. See GraphDB’s connector authenticator examples for more information and example projects that implement such a custom class.

There is no explicitly configurable support for other authentication mechanism supported by Kafka. It should be possible to configure most of them by supplying the relevant Kafka producer properties using the *kafkaProducerConfig* parameter.

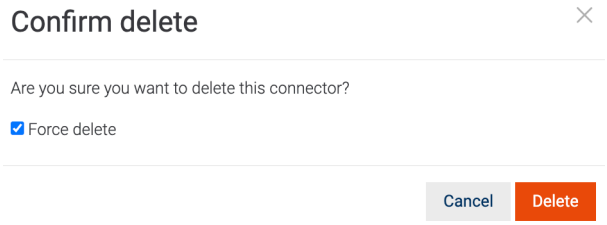
4.7.2.3 Dropping a connector instance

Dropping a connector instance removes all references to its external store from GraphDB as well as the Kafka index associated with it.

The drop command is triggered by a SPARQL `INSERT` with the name of the connector instance in the subject position and `kafka:dropConnector` as the predicate. For example, this removes the connector `my_index`:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon 🗑, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.7.2.4 Retrieving the create options for a connector instance

You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

SELECT ?createString {
  kafka-inst:my_index kafka:listOptionValues ?createString .
}
```

4.7.2.5 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* 🔄 or *Delete* 🗑 icons to perform these operations. Click the *Copy* 📋 icon to copy the connector definition query to your clipboard.

With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>

SELECT ?cntUri ?cntStr {
  ?cntUri kafka:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/kafka/instance#my_index`, while `?cntStr` is bound to a string representing the local name such as "my_index" in this example.

4.7.2.6 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>

SELECT ?cntUri ?cntStatus {
  ?cntUri kafka:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.7.3 Working with data

4.7.3.1 Adding, updating, and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` update requests. This is achieved by intercepting all changes in the plugin and determining which Kafka documents need to be updated.

4.7.4 List of creation parameters

The creation parameters define how the `kafka:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object, where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the *Create Connector user interface* without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to make a Kafka index from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `kafka:graph`, where the prefix `kafka:` represents the IRI `http://www.ontotext.com/connectors/kafka#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *read-only mode*.

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
INSERT {
    GRAPH kafka:graph {
        ...
    }
} WHERE {
        ...
};
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>
INSERT DATA {
    kafka-inst:my_index kafka:createConnector '''
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponding to a single predicate with a field name the same as the predicate, so you must use escaping when issuing Kafka queries.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`kafkaNode` (string), required, the Kafka instance to sync to As Kafka is a third-party service, you have to specify the node where it is running. The node value takes the form `http://hostname.domain:port` or `https://hostname.domain:port`. No default value. Can be updated at runtime with no need to rebuild the index.

`kafkaTopic` (string), required, the Kafka topic to send documents to. No default value.

`kafkaSSL` (boolean), optional, controls whether to use an SSL connection to the Kafka broker. `False` by default. Can be updated at runtime with no need to rebuild the index.

`kafkaPlainAuthUsername` (string), optional, supplies the username for Kafka SASL PLAIN authentication. No default value. Can be updated at runtime with no need to rebuild the index.

`kafkaPlainAuthPassword` (string), optional, supplies the password for Kafka SASL PLAIN authentication. No default value. Can be updated at runtime with no need to rebuild the index.

`kafkaUseTransaction` (boolean), optional, controls whether to use a kafka transaction on entity updates. `true` by default. Can be updated at runtime with no need to rebuild the index. When false, entities go to

kafka immediately regardless of the GraphDB transaction that triggers the update.

`bulkUpdateBatchSize` (integer), controls the maximum batch size in bytes and corresponds to Kafka producer config `batch.`
Default value is `1,048,576` (1 megabyte). Can be updated at runtime with no need to rebuild the index.

`bulkUpdateRequestSize` (integer), controls the maximum request size (and consequently the maximum size per document) i
Default value is `1,048,576` (1 megabyte). Can be updated at runtime with no need to rebuild the index.

`authenticationConfiguratorClass` provides custom authentication behavior Optional.

`kafkaCompressionType` (string), sets the compression to use when sending documents to Kafka. One of `none`, `gzip`, `lz4`, `snappy`), the default is `snappy`. This corresponds to Kafka producer config property `compression.type`. Can be updated at runtime with no need to rebuild the index.

`kafkaProducerId` (string), an optional identifier that allows for separate Kafka producers with different options to the same
No default — all instances to the same Kafka broker will use a shared Kafka producer and thus must have the same options. See also *Producer sharing* and *Conflict resolution*.

`kafkaProducerConfig` (JSON), optional, the settings for creating the Kafka producer. This option is passed directly to the Kafka producer when it is instantiated. Each key is a Kafka producer configuration property. Some configuration keys such as `transactional.id` are not allowed here. The following producer configuration defaults are set:

- `transaction.timeout.ms`: 900000 = 15 minutes and equals the default transaction.max.timeout.ms of a kafka broker
- `batch.size`: 1024 * 1024 = 1MB
- `max.request.size`: 1024 * 1024 = 1MB
- `linger.ms`: 50
- `ack`: all

Can be updated at runtime with no need to rebuild the index.

`kafkaIgnoreDeleteAll` (boolean), optional, a flag that, when selected, will not notify Kafka when all repository statements a
GraphDB handles the removal of all statements as a special operation that is manifested as sending a Kafka record with NULL key and NULL value. If this flag is `true`, no such record will be sent. `False` by default.

`kafkaPropagateConfig` (boolean), optional, a non-persisted flag that, when selected, will force propagation of the Kafka conf
`False` by default. See also *Producer sharing* and *Conflict resolution*. Can be updated at runtime with no need to rebuild the index.

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual, but only some of the languages represented in the literal values can be mapped. You can do this by specifying a list of language ranges to be matched to the language tags of literals according to RFC 4647, Section 3.3.1. Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to Kafka The fields specify exactly which parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in Kafka. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

- **`fieldName` (string), required, the name of the field in Kafka** The name of the field defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field name is used as the key in the JSON document that will be sent to Kafka.

- **`fieldNameTransform` (one of `none`, `predicate`, or `predicate.localName`), optional, `none` by default** Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

  See *Indexing all literals in distinct fields* for an example.
- **`propertyChain` (list of IRIs), required, defines the property chain to reach the value** The property chain defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple, etc. In this model, a property chain with a single element corresponds to a direct property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

  The IRI of the document will be synchronized as the key in the Kafka record.

  See *Copy fields* for defining multiple fields with the same property chain.

  See *Multiple property chains per field* for defining a field whose values are populated from more than one property chain.

  See *Indexing language tags* for defining a field whose values are populated with the language tags of literals.

  See *Indexing the IRI of an entity* for defining a field whose values are populated with the IRI of the indexed entity.

  See *Wildcard literal indexing* for defining a field whose values are populated with literals regardless of their predicate.
- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.
- **`documentFilter` (string), optional, specifies the nested document filter for the field** Only for fields that define nested documents. See also *Entity filtering*.
- **`defaultValue` (string), optional, specifies a default value for the field** Provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or a IRI. It has no default value.
- **`indexed` (boolean), optional, default `true`** If indexed, a field will be included in the JSON document sent to Kafka. `true` by default.

  If `true`, this option corresponds to `"index" = true`. If `false`, it corresponds to `"index" = false`.
- **`multivalued` (boolean), optional, default `true`** RDF properties and synchronized fields may have more than one value. If `multivalued` is set to `true`, all values will be synchronized to Kafka. If set to `false`, only a single value will be synchronized. `true` by default.
- **`ignoreInvalidValues` (boolean), optional, default `false`** Per-field option that controls what happens when a value cannot be converted to the requested (or previously detected) type. `False` by default.

  Example use: when an invalid date literal like `"2021-02-29"^^xsd:date` (2021 is not a leap year) needs to be indexed as a date, or when an IRI needs to be indexed as a number.

  Note: Some conversions are always valid: any literal to an FTS field, any non-literal (IRI, blank node, embedded triple) to a non-analyzed field. When `true`, such values will be skipped with a

note in the logs. When `false`, such values will break the transaction.

- **`array` (boolean), optional, default `false`** Normally, Kafka creates an array only if more than value is present for a given field. If `array` is set to `true`, Kafka will always create an array even for single values. If set to `false`, Kafka will create arrays for multiple values only. `False` by default.
- **`datatype` (string), optional, the manual datatype override** By default, the Kafka GraphDB Connector uses datatype of literal values to determine how they should be mapped to Kafka types. For more information on the supported datatypes, see *Datatype mapping*.

  You can override the datatype mapping with the `"datatype"` property, which can be specified per field. The value of `datatype` can be any of the `xsd:` types supported by the automatic mapping or a native Kafka type prefixed by `native:`, e.g., both `xsd:long` and `native:long` map to the long type in Kafka.
- **`objectFields` (objects array), optional, nested object mapping** When `native:object` is used as a datatype value, this provides a mapping for the nested object's fields. If `datatype` is not provided, then `native:object` will be assumed.

  Nested objects support further nested objects with a limit of five levels of nesting.
- **`startFromParent` (integer), optional, default `0`** Start processing the property chain from the N-th parent instead of the root of the current nested object. 0 is the root of the current nested object, 1 is the parent of the nested object, 2 is the parent of the parent and so on.

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

4.7.4.1 Updating parameters at runtime

As mentioned above, you can update the following connection parameters at runtime with no need to rebuild the index:

- `kafkaNode`
- `kafkaSSL`
- `kafkaProducerConfig`
- `kafkaCompressionType`
- `kafkaPlainAuthUsername`
- `kafkaPlainAuthPassword`
- `kafkaUseTransaction`
- `bulkUpdateBatchSize`
- `kafkaPropagateConfig`

The following SPARQL update request demonstrates this by updating the user and password values:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>
INSERT DATA {
kafka-inst:my_index kafka:updateConnector '''
  {
        "kafkaPlainAuthUsername": "foo"
        "kafkaPlainAuthPassword": "bar"
  }
''' .
}
```

4.7.4.2 Special field definitions

Nested objects

Nested objects are JSON objects that are used as values in the main document or in other nested objects. Up to five levels of nesting is possible. They are defined with the *objectFields* option.

Let's look at an example that uses the following data, which consists of children and grandchildren relations:

```
<urn:John>
    a <urn:Person> ;
    <urn:name> "John" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 60 ;
    <urn:hasSpouse> <urn:Mary> ;
    <urn:hasChild> <urn:Billy> ;
    <urn:hasChild> <urn:Annie> .

<urn:Mary>
    a <urn:Person> ;
    <urn:name> "Mary" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 58 ;
    <urn:hasSpouse> <urn:John> ;
    <urn:hasChild> <urn:Billy> .

<urn:Eva>
    a <urn:Person> ;
    <urn:name> "Eva" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 45 ;
    <urn:hasChild> <urn:Annie> .

<urn:Billy>
    a <urn:Person> ;
    <urn:name> "Billy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 35 ;
    <urn:hasChild> <urn:Tylor> ;
    <urn:hasChild> <urn:Melody> .

<urn:Annie>
    a <urn:Person> ;
    <urn:name> "Annie" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 28 ;
    <urn:hasChild> <urn:Sammy> .

<urn:Tylor>
    a <urn:Person> ;
    <urn:name> "Tylor" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 5 .

<urn:Melody>
    a <urn:Person> ;
    <urn:name> "Melody" ;
    <urn:gender> <urn:Female> ;
    <urn:age> 2 .

<urn:Sammy>
    a <urn:Person> ;
```

(continues on next page)

(continued from previous page)

```
    <urn:name> "Sammy" ;
    <urn:gender> <urn:Male> ;
    <urn:age> 10 .

<urn:Male> <urn:label> "male" .

<urn:Female> <urn:label> "female" .
```

We can create a nested objects index that consists of children and grandchildren with their corresponding fields defining their gender and age:

```
{
  "fields": [
    {
      "fieldName": "name",
      "propertyChain": [
        "urn:name"
      ]
    },
    {
      "fieldName": "age",
      "propertyChain": [
        "urn:age"
      ],
      "datatype": "xsd:long"
    },
    {
      "fieldName": "hasSpouse",
      "propertyChain": [
        "urn:hasSpouse"
      ]
    },
    {
      "fieldName": "gender",
      "propertyChain": [
        "urn:gender",
        "urn:label"
      ]
    },
    {
      "fieldName": "children",
      "propertyChain": [
        "urn:hasChild"
      ],
      "datatype": "native:object",
      "objectFields": [
        {
          "fieldName": "id",
          "propertyChain": [
            "$self"
          ]
        },
        {
          "fieldName": "name",
          "propertyChain": [
            "urn:name"
          ]
        },
        {
          "fieldName": "age",
          "propertyChain": [
```

(continues on next page)

(continued from previous page)

```
          "urn:age"
        ],
        "datatype": "xsd:long"
      },
      {
        "fieldName": "gender",
        "propertyChain": [
          "urn:gender",
          "urn:label"
        ]
      },
      {
        "fieldName": "children",
        "propertyChain": [
          "urn:hasChild"
        ],
        "objectFields": [
          {
            "fieldName": "id",
            "propertyChain": [
              "$self"
            ]
          },
          {
            "fieldName": "name",
            "propertyChain": [
              "urn:name"
            ]
          },
          {
            "fieldName": "age",
            "propertyChain": [
              "urn:age"
            ],
            "datatype": "xsd:long"
          }
        ]
      }
    ]
  },
  {
    "fieldName": "grandChildren",
    "valueFilter": "$this -> type in (<urn:Person>)",
    "propertyChain": [
      "urn:hasChild",
      "urn:hasChild"
    ],
    "datatype": "native:object",
    "objectFields": [
      {
        "fieldName": "id",
        "propertyChain": [
          "$self"
        ]
      },
      {
        "fieldName": "name",
        "propertyChain": [
          "urn:name"
        ]
      },
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "age",
          "propertyChain": [
            "urn:age"
          ],
          "datatype": "xsd:long"
        },
        {
          "fieldName": "gender",
          "propertyChain": [
            "urn:gender",
            "urn:label"
          ]
        }
      ]
    },
  ],
  "types": [
    "urn:Person"
  ],
  "kafkaNode": ...,
  "kafkaTopic": ...
}
```

Copy fields

It is often convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases. The Kafka GraphDB Connector has explicit support for fields that copy their value from another field. To do this, specify a single element in the property chain of the form `@otherFieldName`, where `otherFieldName` is another non-copy field. Take the following example:

```
...
  "fields": [
    {
      "fieldName": "grape",
      "facet": false,
      "propertyChain": [
        "http://www.ontotext.com/example/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
    {
      "fieldName": "whiteGrape",
      "propertyChain": [
        "@grape"
      ]
    }
  ],
  "entityFilter": "?whiteGrape -> type = <wine:WhiteGrape>"
...
```

The snippet creates a field “grape” containing all grapes, and another field “whiteGrape”. Both fields are populated with the same values initially and “whiteGrape” is defined as a copy field that refers to the field “grape”. The field “whiteGrape” is additionally filtered so that only certain grape varieties will be synchronized.

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in Kafka) with more than one property chain — for example, the concept of "name" could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in Kafka, you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.example.com/demo#canonicalName"
      ],
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.example.com/demo#historicalName"
      ]
      ...
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in Kafka.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old`, you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:createConnector '''
```

(continues on next page)

(continued from previous page)

```
    {
      "kafkaNode": "localhost:9092",
      "kafkaTopic": "my_index",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "lang()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.example.com/demo#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:createConnector '''
    {
      "kafkaNode": "localhost:9092",
      "kafkaTopic": "my_index",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "graph()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.example.com/`

`demo#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:createConnector '''
    {
      "kafkaNode": "localhost:9092",
      "kafkaTopic": "my_index",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "feature",
           "propertyChain": [
             "http://www.example.com/demo#feature",
             "localName()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.example.com/demo#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
    kafka-inst:my_index kafka:createConnector '''
{
  "kafkaNode": "localhost:9092",
  "kafkaTopic": "my_index",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "entityId",
      "propertyChain": [
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

Note: Note that GraphDB will also use the IRI of each entity as the ID of each document in Kafka, which is represented by the field `id`.

4.7.5 Datatype mapping

The Kafka GraphDB Connector maps different types of RDF values to different types of Kafka values according to the basic type of the RDF value (IRI or literal) and the datatype of literals. The auto-detection uses the following mapping:

RDF value	RDF datatype	JSON type
IRI	n/a	`string`
literal	any type not explicitly mentioned below	`string`
literal	`xsd:boolean`	`boolean`
literal	`xsd:double`	`number`
literal	`xsd:float`	`number`
literal	`xsd:long`	`number`
literal	`xsd:int`	`number`
literal	`xsd:dateTime`	`string` in ISO 8601 format with time zone
literal	`xsd:date`	`string` in ISO format without time zone
literal	`xsd:time`	`string` in ISO format with time zone
literal	`xsd:gYear`	`string` in ISO format without time zone
literal	`xsd:gYearMonth`	`string` in ISO format without time zone

Note: For any given field, the automatic mapping uses the first value it sees. This works fine for clean datasets but might lead to problems if your dataset has non-normalized data — for example, the first value has no datatype but other values do.

Because of this, it's best to set the *datatype* to a fixed value such as `xsd:date`.

Please note that the commonly used `xsd:integer` and `xsd:decimal` datatypes are not indexed as numbers because they represent infinite precision numbers. You can override that by using the *datatype* option to cast them to `xsd:long`, `xsd:double`, or `xsd:float` as appropriate.

4.7.5.1 Date and time conversion

RDF and ISO use slightly different models for representing dates and times, even though the values might look very similar.

Years in RDF values use the XSD format and are *era* years, where positive values denote the common era and negative values denote years before the common era. There is no year zero.

Years in the ISO format are *proleptic* years: positive values denote years from the common era with any previous eras just going down by one mathematically so there is year zero.

In short:

- year 2020 CE = year 2020 in XSD = year 2020 in ISO.
- …
- year 1 CE = year 1 in XSD = year 1 in ISO.
- year 1 BCE = year -1 in XSD = year 0 in ISO.
- year 2 BCE = year -2 in XSD = year -1 in ISO.
- …

All years coming from RDF literals will be converted to ISO before sending to Kafka.

Both XSD and ISO date and time values support timezones. In addition to that, XSD defines the lack of a timezone as *undetermined*. Since we do not want to have any undetermined state in the indexing system, GraphDB defines the undetermined time zone as UTC. For example, `"2020-02-14T12:00:00"^^xsd:dateTime` is equivalent to `"2020-02-14T12:00:00Z"^^xsd:dateTime` (Z is the UTC timezone, also known as +00:00).

Also note that XSD dates and partial dates, e.g., `xsd:gYear` values, may have a timezone, which leads to additional complications. For example, `"2020+02:00"^^xsd:gYear` (the year 2020 in the +02:00 timezone) will be normalized to `2019-12-31T22:00:00Z` (the previous year!) if GraphDB followed strict timezone adherence. We have chosen to ignore the timezone on any values that do not have an associated time value. For example, all of the following values will be treated as if they specified UTC as their timezone:

- `"2020-02-15+02:00"^^xsd:date`
- `"2020-02+02:00"^^xsd:gYearMonth`
- `"2020+02:00"^^xsd:gYear`

4.7.6 Entity filtering

The Kafka connector supports four kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to Kafka if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable.

4.7.6.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

On nested documents, the per-field value filter can be used to remove the entire nested document early in the indexing process, e.g., by checking the type of the nested document via next hop with `rdf:type`.

Nested document filter The nested document filter is specified with the `documentFilter` parameter **inside the field definition** of the field that defines the root of a nested document. The filter is evaluated after the entire nested document has been collected. Failing to pass this filter removes the entire nested document.

Inside a nested document filter, the field names are within the context of the nested document and not within the context of the top-level document. For example, if you have a `children` field that defines a nested document and you use a filter like `?age < "10"^^xsd:int`, it will refer to the field `children.age`. You can use the prefix `$outer.` one or more times to refer to field values from the outer document from the viewpoint of the nested document. For example, `$outer.age > "25"^^xsd:int` will refer to the `age` field that is a sibling of the `children` field.

Other than the above differences, the nested document filter is equivalent to the top-level document filter from the viewpoint of the nested document.

See also *Migrating from GraphDB 9.x*.

4.7.6.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on original RDF values before they are converted to Kafka values using *datatype mapping*.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

Continued on next page

Table 14 – continued from previous page

Operator	Meaning

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible such as `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height` has a value of `"150"^^xsd:int` and `dateOfBirth` has a
	value of `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`

Continued on next page

Table 14 – continued from previous page

Operator	Meaning
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.7.6.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.7.6.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter and the nested document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.7.6.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.example.com/demo#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:createConnector '''
    {
      "kafkaNode": "localhost:9092",
      "kafkaTopic": "my_index",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "name",
          "propertyChain": ["http://www.example.com/demo#name"]
        },
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo#city"],
          "valueFilter": "$this = \\"London\\""
        }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo#city"],
          "defaultValue": "London"
        }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.7.6.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.example.com/demo2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
    rdf:type example2:Person ;
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
```

(continues on next page)

(continued from previous page)

```
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to Kafka, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>
PREFIX kafka-inst: <http://www.ontotext.com/connectors/kafka/instance#>

INSERT DATA {
  kafka-inst:my_index kafka:createConnector '''
    {
      "kafkaNode": "localhost:9092",
      "kafkaTopic": "my_index",
      "types": ["http://www.example.com/demo2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
         {
           "fieldName": "taggedWithPerson",
           "propertyChain": ["http://www.example.com/demo2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo2#Person>"
         },
         {
           "fieldName": "taggedWithLocation",
           "propertyChain": ["http://www.example.com/demo2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo2#Location>"
         }
      ]
    }
  ''' .
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

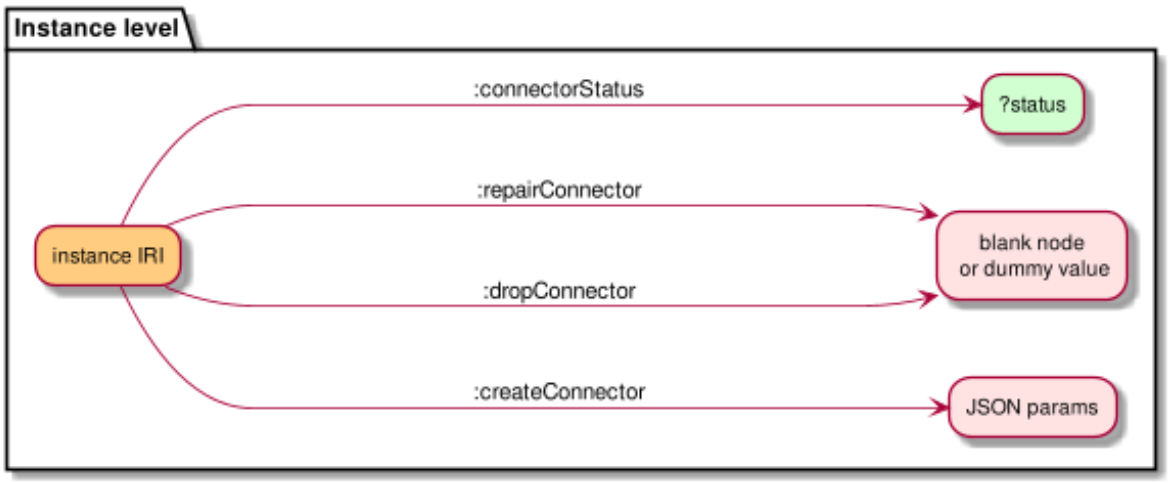
The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

4.7.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.7.8 Caveats

4.7.8.1 Producer sharing

The Kafka connector aims to minimize resource usage and provide smooth transactional operation. It does this by using a single Kafka producer object for each connector instance that is connected to the same Kafka broker node. This has the following benefits:

- It uses less memory because each Kafka producer requires a certain amount of buffer memory.
- A failed transaction in one Kafka connector instance will be reverted in all other Kafka connector instances together with the GraphDB transaction.

Due to the nature of Kafka producers it imposes a restriction as well:

- All connector instances must use the same Kafka options. For example, they must have the same values for the `bulkUpdateBatchSize` and `kafkaCompressionType` options.

Once you have created at least one Kafka connector instance and attempted to create another instance, the following are possible scenarios:

Different Kafka broker

- The new connector instance specifies a different Kafka broker.
- The connector instance will be created and a new Kafka producer will be instantiated.

Same Kafka broker + same Kafka options

- The new connector instance specifies the same Kafka broker as one of the existing connectors and the *same* options as the existing connector.
- The connector instance will be created and the existing Kafka producer will be reused.

Same Kafka broker + different Kafka options

- The new connector instance specifies the same Kafka broker as one of the existing connectors and *different* options than the existing connector.
- The connector instance will *not* be created and an error explaining the reason will be thrown.
- See *Conflict resolution* for possible workarounds.

Note: The Kafka broker for two connector instances is considered to be the same if at least one of the host-port pairs supplied via the `kafkaNode` option is the same.

Conflict resolution

If the attempt to create a new Kafka connector instance was denied because another instance was already created with different options, there are several possible ways to resolve the conflict:

Manual resolution

- Examine the options of the new connector instance you want to create.
- Make the options the same as of the existing connector instance.

Propagate the new options to the existing instances

- Set the option *kafkaPropagateConfig* of the new instance to `true`.
- The new options will be propagated to all existing instances that share the same Kafka broker node.

Force the allocation of a new producer

- Set the option *kafkaProducerId* of the new instance to some non-empty identifier.

- This will override the producer sharing mechanism and allocate a new producer associating it with the supplied producer ID.
- The new connector will use the new options.
- All existing instances will continue using their previous options.

4.7.9 Upgrading from previous versions

4.7.9.1 Migrating from GraphDB 9.x

GraphDB 10.0 introduces major changes to the filtering mechanism of the connectors. Existing connector instances will not be usable and attempting to use them for queries or updates will throw an error.

If your GraphDB 9.x (or older) connector definitions **do not** include an entity filter, you can simply *repair them*.

If your GraphDB 9.x (or older) connector definitions **do** include an entity filter with the `entityFilter` option, you need to rewrite the filter with one of the current *filter types*:

1. Save your existing connector definition.
2. Drop the connector instance.
3. In general, most older connector filters can be easily rewritten using the per-field value filter and top-level document filter. Rewrite the filters as follows.

   Rule of thumb:

   - If you want to remove individual values, i.e., if the operand *is not* `BOUND()` –> rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand *is* `BOUND()` –> rewrite with top-level document filter.

   So if we take the example:

   ```
   ?location = <urn:Europe> AND BOUND(?location) AND ?type IN (<urn:Foo>, <urn:Bar>)
   ```

   It needs to be rewritten like this:

   - Per-field rule on field `location`: `$this = <urn:Europe>`
   - Per-field rule on field `type`: `$this IN (<urn:Foo>, <urn:Bar>)`
   - Top-level document filter: `BOUND(?location)`
4. Recreate the connector instance using the new definition.

4.8 Kafka Sink Connector

Note: Despite having a similar name, the Kafka Sink connector is not a *GraphDB connector*.

Modern businesses have a growing need to integrate data from multiple diverse systems. Automating the update and continuous build of knowledge graphs with the incoming streams of data can be cumbersome due to a number of reasons: verbose functional code writing, numerous transactions per update, suboptimal usability of GraphDB's RDF mapping language, and the lack of a direct way to stream updates to knowledge graphs.

GraphDB's open-source Kafka Sink connector, which supports *smart updates* with SPARQL templates, solves this issue by reducing the amount of code needed for raw event data transformation, making it easier to automate knowledge graph updates. It is a separately running process, which helps to avoid database sizing. The connector allows for customization according to the user's specific business logic and requires no GraphDB downtime during configuration.

With it, users can push update messages to Kafka, after which a Kafka consumer processes them and applies the updates in GraphDB.

4.8.1 Setup

Important: Before setting up the connector, make sure to have JDK 11 installed.

1. Install Kafka 3.8.0 or newer.
2. The Kafka Sink connector can be deployed in a Docker container. To install it and verify that it is working correctly, follow the GitHub README instructions.

4.8.2 Update types

The Kafka Sink connector supports three types of updates:

- simple add
- replace graph
- smart update with a *DELETE/INSERT template*

A given Kafka topic is configured to accept updates in a predefined mode and format. The format must be one of the *supported RDF formats*.

4.8.2.1 Simple add

This is a simple `INSERT` operation where no document identifiers are needed and new data is always added as is. All you need to provide is the new RDF data to add. The following is valid:

- The Kafka topic is configured to only add data.
- The Kafka key is irrelevant but it is recommended to use a unique ID such as a random UUID.
- The Kafka value is the new RDF data to add.

Let's look at an example scenario.

1. Start GraphDB on the same or a different machine.
2. In GraphDB, create a repository called "kafka-test".
3. To deploy the connector, execute in the project's `docker-compose` directory:

```
sudo docker-compose up --scale graphdb=0
```

   where `graphdb=0` denotes that GraphDB must be started outside of the Docker container.

4. Next, configure the Kafka sink connector that will add data into the repository. In the Kafka sink connector's directory, execute this `curl` command:

```
curl http://localhost:8083/connectors \
    -H 'Content-Type: application/json' \
    --data '{"name":"kafka-sink-graphdb-add",
        "config":{
         "graphdb.server.url":"http://graphdb.example.com:7200",
         "connector.class":"com.ontotext.kafka.GraphDBSinkConnector",
         "key.converter":"com.ontotext.kafka.convert.DirectRDFConverter",
         "value.converter":"com.ontotext.kafka.convert.DirectRDFConverter",
         "value.converter.schemas.enable":"false",
         "topics":"gdb-add",
```

(continues on next page)

(continued from previous page)

```
        "tasks.max":1,
        "offset.storage.file.filename":"/tmp/storage-add",
        "graphdb.server.repository":"kafka-test",
        "graphdb.batch.size":64,
        "graphdb.batch.commit.limit.ms":1000,
        "graphdb.auth.type":"NONE",
        "graphdb.update.type":"ADD",
        "graphdb.update.rdf.format":"nq"}}'
```

This `curl` command includes the following parameters:

- `topics`, which can be one of the following:
  - the name of the topic from which the connector will be reading the documents; in this example, `gdb-add`
  - a comma-separated list of topics
- `graphdb.server.url`: the URL of the GraphDB server; adjust the sample value `http://graphdb.example.com:7200` to reflect your server's location

  Important: Since GraphDB is running outside the Kafka Sink Docker container, referencing `localhost` in `graphdb.server.url` will not work. Use a hostname or IP address that is visible from within the container.

- `graphdb.server.repository`: the GraphDB repository in which the connector will write the documents; in the example above, `kafka-test`
- `graphdb.update.type`: the type of the update; `ADD` above

This will create the add data connector, which will read documents from the `gdb-add` topic and send it to the `kafka-test` repository on the respective GraphDB server.

5. For the purposes of the example, we will also create a test Kafka producer that will write in the respective Kafka topic. In the Kafka installation directory, execute the following command:

```
bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-add
```

6. That command will then be waiting for data via standard input, so paste the following RDF data in the same window and press `Enter`:

```
<http://example/subject> <http://example/predicate> "This is an example of adding data"
↪<http://example/graph> .
```

7. To finish the data input press `Ctrl+Z` if you're using Windows and `Ctrl/Cmd+D` for any other operating system.
8. In the Workbench SPARQL editor of the "kafka-test" repository, run the query:

```
SELECT * WHERE {
    GRAPH ?g {
        ?s ?p ?o
    }
}
```

9. The RDF data that we just added via the producer should be returned as result.

4.8.2.2 Replace graph

In this update type, a document (the smallest update unit) is defined as the contents of a named graph. To perform an update, the following information must be provided:

- The IRI of the named graph — the document ID
- The new RDF contents of the named graph — the document contents

The update is performed with these steps:

- The Kafka topic is configured for replace graph.
- The Kafka key defines the named graph to update.
- The Kafka value defines the contents of the named graph.

Let's look at an example:

1. After following the steps above, the Docker container with the Kafka sink connector should already be running, and the "kafka-test" repository will have been created.
2. Now, let's configure the Kafka sink connector that will replace data in a named graph. In the Kafka sink connector's directory, execute this `curl` command:

```
curl http://localhost:8084/connectors \
      -H 'Content-Type: application/json'\
     --data '{"name":"kafka-sink-graphdb-replace",
           "config":{
              "graphdb.server.url":"http://graphdb.example.com:7200",
              "connector.class":"com.ontotext.kafka.GraphDBSinkConnector",
              "key.converter":"com.ontotext.kafka.convert.DirectRDFConverter",
              "value.converter":"com.ontotext.kafka.convert.DirectRDFConverter",
              "value.converter.schemas.enable":"false",
              "topics":"gdb-replace",
              "tasks.max":1,
              "offset.storage.file.filename":"/tmp/storage-replace",
              "graphdb.server.repository":"kafka-test",
              "graphdb.batch.size":64,
              "graphdb.batch.commit.limit.ms":1000,
              "graphdb.auth.type":"NONE",
              "graphdb.update.type":"REPLACE_GRAPH",
              "graphdb.update.rdf.format":"nq"}}'
```

   This uses the same important parameters as in the add data example above.

   This will configure the replace graph connector, which will read data from the `gdb-replace` topic and send it to the `kafka-test` repository on the respective GraphDB server.

3. To replace data in a specific named graph, you need to provide:

   - the name of the graph as the key of the Kafka message in the Kafka topic
   - the new data for the replace operation as the value of the Kafka message

   So, you need to modify the producer to create a key-value message. In the Kafka installation directory, execute the following command:

```
bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-replace \
    --property parse.key=true --property key.separator="-"
```

4. To replace the data in the graph, paste the following into the same window and then press `Enter` and either `Ctrl+Z` or `Ctrl/Cmd+D` like before to show that you are done entering data:

```
http://example/graph-<http://example/subject> <http://example/predicate> "Successfully
↪replaced graph" <http://example/graph> .
```

   The key value must be the `?id` value from the template.

5. To see the replaced data, run the query from above in the Workbench SPARQL editor of the "kafka-test" repository:

```
SELECT * WHERE {
    GRAPH ?g {
        ?s ?p ?o
    }
}
```

6. The replaced data in the graph should be returned as result.

4.8.2.3 DELETE/INSERT template

In this update type, a document is defined as all triples for a given document identifier according to a predefined schema. The schema is described as a *SPARQL DELETE/INSERT template* that can be filled from the provided data at update time. The following must be present at update time:

- The SPARQL template update, which must be predefined, not provided at update time
  - this can be a `DELETE WHERE` update that only deletes the previous version of the document and the new data is inserted as is
  - this can be a `DELETE INSERT WHERE` update that deletes the previous version of the document and adds additional triples such as timestamp information
- The IRI of the updated document
- The new RDF contents of the updated document

The update is performed with these steps:

- The Kafka topic is configured for a specific template.
- The Kafka value defines the data to be updated for the bound ID.

Important: One instance configuration can work with one template. If you need to use a second template, you can create a second configuration on the same instance.

Let's look at an example scenario:

1. As with the previous update, the Docker container with the Kafka Sink connector running is already created, and the "kafka-test" repository is used for the purposes of the example.
2. With this type of update, you need to first create a SPARQL template for the data update.
   a. In Workbench, go to *Setup* ▸ *SPARQL Templates* ▸ *Create new SPARQL template*.
   b. Enter a template IRI (required) such as `http://example.com/my-template`.
   c. Go to *Import* ▸ *User data* ▸ *Import RDF text snippet* and import the following `sample data` describing five fictitious wines:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine .
wine:WhiteWine rdfs:subClassOf wine:Wine .
wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .
```

(continues on next page)

(continued from previous page)

```
wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer .
```

d. Next, navigate to *Setup* ▸ *SPARQL Templates* ▸ *Create new SPARQL template* to create the SPARQL template:

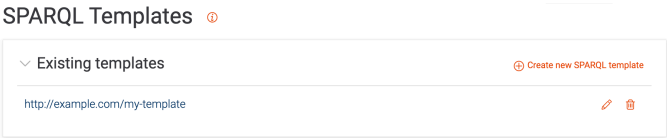
```
PREFIX wine: <http://www.ontotext.com/example/wine#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

DELETE {
    ?s wine:hasSugar ?oldValue .
    ?s wine:hasYear ?oldYear
} INSERT {
    ?s wine:hasSugar ?sugar .
    ?s wine:hasYear ?year .
} WHERE {
```

(continues on next page)

(continued from previous page)

```
    ?s ?p ?oldValue .
    ?s ?p1 ?oldYear .
}
```

e. *Save* it, after which it will appear in the templates list.



Now you need to configure the Kafka sink connector that will update some data in a named graph. In the directory of the Kafka sink connector, execute:

```
curl http://localhost:8085/connectors \
    -H 'Content-Type: application/json' \
    --data '{"name": "kafka-sink-graphdb-update",
        "config": {
        "graphdb.server.url":"http://graphdb.example.com:7200",
        "connector.class": "com.ontotext.kafka.GraphDBSinkConnector",
        "key.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
        "value.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
        "value.converter.schemas.enable": "false",
        "topics": "gdb-update",
        "tasks.max": 1,
        "offset.storage.file.filename": "/tmp/storage-update",
        "graphdb.server.repository": "kafka-test",
        "graphdb.batch.size": 64,
        "graphdb.batch.commit.limit.ms": 1000,
        "graphdb.auth.type": "NONE",
        "graphdb.update.type": "SMART_UPDATE",
        "graphdb.update.rdf.format": "jsonld",
        "graphdb.template.id":
        "http://example.com/my-template"}}'
```

Note how the data passed by the `curl` command specifies the template IRI `"graphdb.template.id":"http://example.com/my-template"` that was created in Workbench earlier, which the sink connector will execute.

1. In the Kafka installation directory, execute the following command:

```
bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-update
```

2. To execute an update, paste the following into the same window and then press `Enter` and either `Ctrl+Z` or `Ctrl/Cmd+D` like before to show that you are done entering data:

```
{"sugar" : "none" , "year" : 2020 , "s" : "http://www.ontotext.com/example/wine
↪#Blanquito" }
```

3. To see the updated data, run the query from above in the Workbench SPARQL editor of the "kafka-test" repository:

```
PREFIX wine: <http://www.ontotext.com/example/wine#>

SELECT ?s ?p ?o
WHERE {
   BIND(wine:Blanquito as ?s ) .
    ?s ?p ?o .
}
```

4. The updated data in the graph should be returned as result.

4.8.3 Configuration properties

The following properties are used to configure the Kafka Sink connector:

Property	Description	Valid values
`name`	Globally unique name to use for this connector.	String
`connector.class`	Name or alias of the class for this connector. Must be a subclass of `org.apache.kafka.connect.connector.Connector`. If the connector is `org.apache.kafka.connect.file.FileStreamSinkConnector`, you can either specify this full name or use `FileStreamSink` or `FileStreamSinkConnector` to make the configuration a bit shorter.	String
`key.converter`	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the keys in messages written to or read from Kafka, and since this is independent of connectors, it allows any connector to work with any serialization format. Default is `NULL`.	Class
`value.converter`	Converter class used to convert between Kafka Connect format and the serialized form that is written to Kafka. This controls the format of the values (as opposed to keys) in messages written to or read from Kafka, and since this is independent of connectors, it allows any connector to work with any serialization format. Default is `NULL`.	Class
`topics`	Comma-separated list of topics to consume.	List
`tasks.max`	Maximum number of tasks to use for this connector. Default is `1`.	Integer
`graphdb.server.url`	The URL of the GraphDB server.	String
`graphdb.server.repository`	The GraphDB repository where the connector will write the documents.	String
`graphdb.batch.size`	The maximum number of documents to send from Kafka to GraphDB in one transaction.	Integer
`graphdb.auth.type`	The authentication type.	`NONE`, `BASIC`, `CUSTOM`
`graphdb.auth.basic.username`	The username for basic authentication.	String
`graphdb.auth.basic.password`	The password for basic authentication.	String
`graphdb.auth.header.token`	The GraphDB authentication token.	String
`graphdb.update.type`	The type of the transaction.	`ADD`, `REPLACE_GRAPH`, `SMART_UPDATE`
`graphdb.update.rdf.format`	The format of the documents sent from Kafka to GraphDB. Default is `ttl`.	Any *supported RDF format* file extension. For example, to send Turtle-star to GraphDB, set this to `ttls`, or to send JSON-LD, set it to `jsonld`.
`graphdb.batch.commit.limit.ms`	The timeout applied per batch that is not full before it is committed. Default is `3000`.	Long
`graphdb.tls.thumbprint`	The SHA-256 thumbprint of the GraphDB certificate that the connector should import and trust.	String
`graphdb.tls.hostname.`	Enables the hostname verification against the certificate or certificate chain. Default is `true`.	Boolean

4.9 ChatGPT Retrieval GraphDB Connector

The ChatGPT Retrieval GraphDB Connector provides a means to convert an RDF model to a text representation and synchronize it to the ChatGPT Retrieval Plugin, which then converts the text document to embedded vectors and indexes it into its configured vector database. This is an experimental feature that is mostly meant to be used together with the *Talk to Your Graph* feature but can also be used independently to query the vector database.

Note: GraphDB supports *full-text search* options as well.

GraphDB's Connectors provide synchronization at the *entity* level, where an entity is defined as having a unique identifier (an IRI) and a set of properties and property values. In terms of RDF, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support *property chains*. A property chain is a sequence of triples where each triple's object is the subject of the following triple.

The main features of the connector are:

- maintenance of an index that is always in sync with the data stored in GraphDB
- multiple independent instances per repository
- entity synchronization defined by:
  - a list of fields on the ChatGPT Retrieval plugin side and property chains on the GraphDB side whose values will be synchronized
  - a list of `rdf:type` values of the entities for synchronization
  - a list of languages for synchronization, with all languages being the default
  - Additional filtering by property and value
- text search via ChatGPT Retrieval plugin queries
- chunk and metadata extraction from search results
- paging of results using `OFFSET` and `LIMIT`

4.9.1 Usage

All interactions with the ChatGPT Retrieval GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- `INSERT` for creating, updating, and deleting **connector instances**
- `SELECT` for listing **connector instances** and querying their configuration parameters
- `INSERT` and `SELECT` for storing and querying **data** as part of the normal GraphDB data workflow

In general, this corresponds to `INSERT` update requests that add or modify data and to `SELECT` queries of existing data.

Each connector implementation uses its own namespace IRI to distinguish it from other connectors. For the ChatGPT Retrieval GraphDB Connector, this is `http://www.ontotext.com/connectors/retrieval#`. Each command or predicate executed by the connector uses terms from this namespace (for example, `http://www.ontotext.com/connectors/retrieval#createConnector`) to create a connector instance for ChatGPT Retrieval.

Individual instances of a connector are distinguished by unique names that are also IRIs. They have their own namespace to avoid clashing with any of the command predicates. For ChatGPT Retrieval, the instance IRI is `http://www.ontotext.com/connectors/retrieval/instance#`.

> **Warning:** Changing the ChatGPT Retrieval Plugin URL will not reindex the data automatically. If you need the data to be reindexed, you can *repair* the connector instance.

Sample data All examples use the following `sample data` that describes five fictitious wines: Yoyowine, Franvino, Noirette, Blanquito, and Rozova, as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix wine: <http://www.example.com/demo/wine#> .

wine:RedWine rdfs:subClassOf wine:Wine ;
   rdfs:label "Red Wine" .
wine:WhiteWine rdfs:subClassOf wine:Wine ;
   rdfs:label "White Wine" .
wine:RoseWine rdfs:subClassOf wine:Wine ;
   rdfs:label "Rose Wine" .

wine:Merlot
    rdf:type wine:Grape ;
    rdfs:label "Merlot" .

wine:CabernetSauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine:CabernetFranc
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Franc" .

wine:PinotNoir
    rdf:type wine:Grape ;
    rdfs:label "Pinot Noir" .

wine:Chardonnay
    rdf:type wine:Grape ;
    rdfs:label "Chardonnay" .

wine:Yoyowine
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:CabernetSauvignon ;
    wine:hasSugar "dry" ;
    wine:hasYear "2013"^^xsd:integer ;
    wine:hasWinery "Semantinos" .

wine:Franvino
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:Merlot ;
    wine:madeFromGrape wine:CabernetFranc ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer ;
    wine:hasWinery "Semantinos" .

wine:Noirette
    rdf:type wine:RedWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2012"^^xsd:integer ;
    wine:hasWinery "In vino veritas" .
```

(continues on next page)

(continued from previous page)

```
wine:Blanquito
    rdf:type wine:WhiteWine ;
    wine:madeFromGrape wine:Chardonnay ;
    wine:hasSugar "dry" ;
    wine:hasYear "2012"^^xsd:integer ;
    wine:hasWinery "In vino veritas" .

wine:Rozova
    rdf:type wine:RoseWine ;
    wine:madeFromGrape wine:PinotNoir ;
    wine:hasSugar "medium" ;
    wine:hasYear "2013"^^xsd:integer ;
    wine:hasWinery "In vino veritas" .
```

4.9.2 Setup and maintenance

4.9.2.1 Setting up the ChatGPT retrieval plugin

The ChatGPT retrieval connector requires a running instance of the ChatGPT Retrieval plugin. Ontotext maintains a fork of the ChatGPT Retrieval Plugin that contains some important fixes and improvements described in the ONTOTEXT.md file of the fork. The fork also provides installation instructions for the Retrieval plugin.

Once you have set up the ChatGPT retrieval plugin, you will need to generate a Bearer token. This is a secret token used to authenticate requests to the plugin API. You can generate one using any tool or method you prefer. For example, you can go to https://jwt.io/#decoded-jwt and paste this into the payload field on the right:

```
{
  "sub": "1234567890",
  "name": "Test",
  "iat": 1694775299
}
```

The token will appear in the token field on the left and will look like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlRlc3QiLCJpYXQiOjE2OTQ3NzUyOTl9.DC7JjYLShRJ7r73btG6tWwPM-
↪VNWEf41Vk1nxFkt0v0
```

4.9.2.2 Creating a connector instance

Create a connector instance by sending a SPARQL query with the following configuration data:

- the name of the connector instance, such as `my_index`
- a ChatGPT Retrieval instance to synchronize to
- classes to synchronize
- properties to synchronize

The configuration data must be provided as a JSON string representation and passed together with the create command.

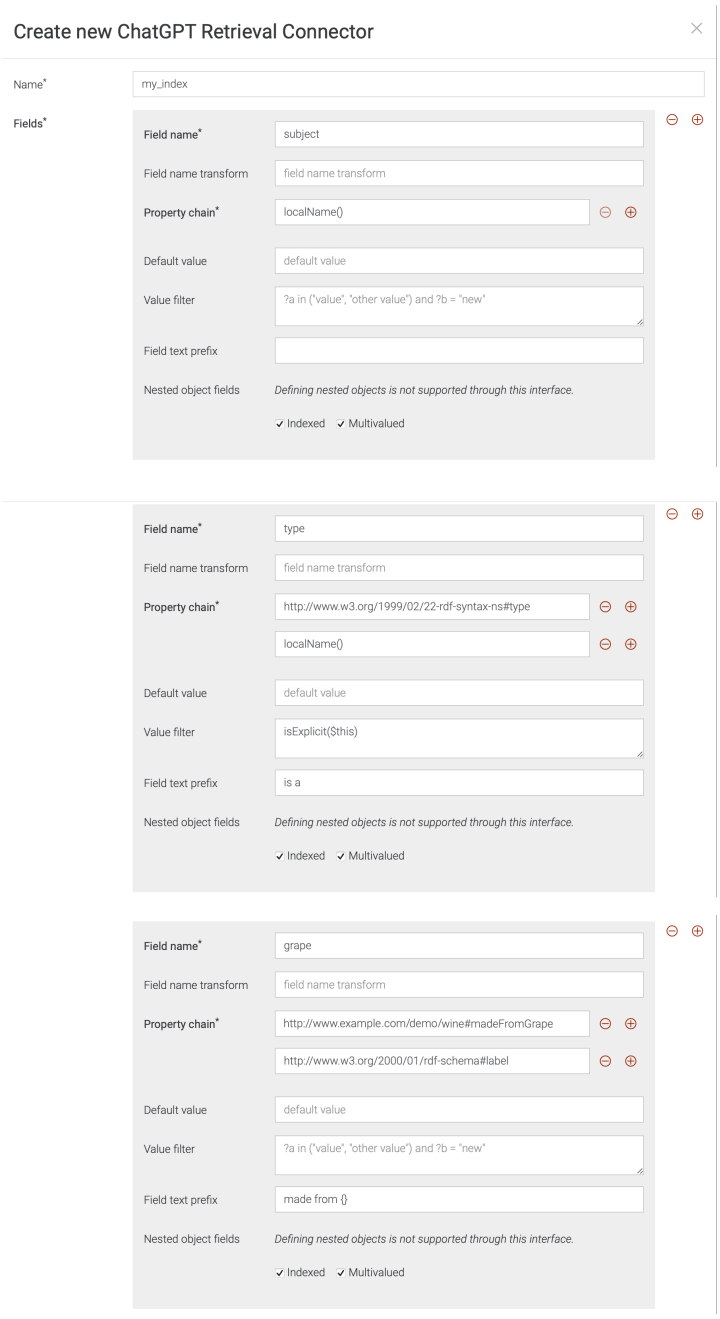
You can create connectors via *Workbench* or by *using a SPARQL update query* (create command).

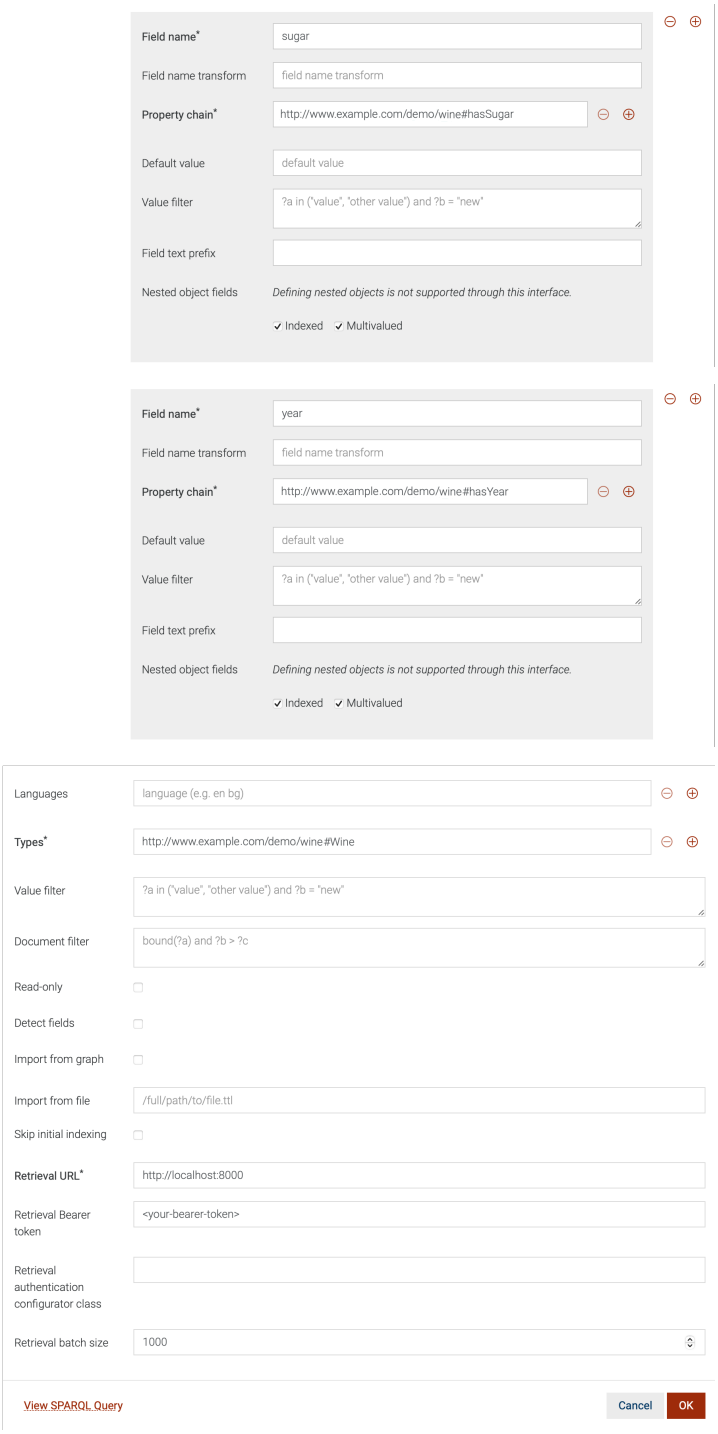
If you create the connector using Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using Workbench

Warning: The Workbench Connector management view does not support the creation of nested fields. While the example in *Using the create command* below has the nested field "metadata", it is missing from the screenshots shown here. Having the "metadata" field is not essential for most of the query examples further below.

1. Go to *Setup* ▸ *Connectors*.
2. Click *New Connector* in the tab of the respective Connector type you want to create.
3. Fill out the configuration form.





4. Execute the `CREATE` statement that will be generated from the data entered on the form by clicking *OK*. You can also view its SPARQL query by clicking *View SPARQL Query* and then copy it to execute it manually or integrate it in automation scripts.

Using the create command

The create command is triggered by a SPARQL `INSERT` with the `retr:createConnector` predicate. In the example, it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above.

To be able to use newlines and quotes in the block of JSON being passed without the need for escaping, the update request uses SPARQL's multi-line string delimiter consisting of 3 apostrophes: `'''...'''`. You can also use 3 quotes instead: `"""..."""`.

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
    retr-index:my_index retr:createConnector '''
{
  "retrievalUrl": "http://localhost:8000",
  "retrievalBearerToken": "<replace-this-with-actual-value>",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "subject",
      "propertyChain": [
        "localName()"
      ]
    },
    {
      "fieldName": "metadata",
      "propertyChain": [
        "$self"
      ],
      "objectFields": [
        {
          "fieldName": "author",
          "propertyChain": [
            "http://www.example.com/demo/wine#hasWinery"
          ]
        }
      ]
    },
    {
      "fieldName": "type",
      "propertyChain": [
        "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
        "localName()"
      ],
      "fieldTextPrefix": "is a",
      "valueFilter": "isExplicit($this)"
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "fieldTextPrefix": "made from {}"
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasSugar"
```

(continues on next page)

(continued from previous page)

```
      ]
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.example.com/demo/wine#hasYear"
      ]
    }
  ]
}
''' .
}
```

The above command creates a new ChatGPT Retrieval connector instance that connects to the ChatGPT Retrieval instance accessible at the `http://localhost:8000` URL.

The `"types"` key defines the RDF type of the entities to synchronize. In the example, it is only entities of the type `http://www.example.com/demo/wine#Wine` and its subtypes if RDFS or higher-level reasoning is enabled. The `"fields"` key defines the mapping from RDF to ChatGPT Retrieval. The basic building block is the property chain: a sequence of RDF properties where the object of each property is the subject of the following property. In the example, three bits of information are mapped — the grape the wines are made of, sugar content, and year. Each chain is assigned a short and convenient field name: "grape", "sugar", and "year".

The field `grape` is an example of a property chain composed of more than one property. First, we take the wine's `madeFromGrape` property, the object of which is an instance of the type Grape, and then we take the `rdfs:label` of this instance. The fields `sugar` and `year` are both composed of a single property that links the value directly to the wine.

The `type` field uses a property chain whose last element is *localName()*. This indicates that values mapped to that field should consist of the local name of the IRI value instead of the entire IRI — in this case, the name of the class that the resource belongs to.

Two *helper field mappings* used by the example above are:

- `subject` maps the values used to construct the beginning of each text document. It uses the `localName()` construct just like the `type` field. In this connector definition, it's simply the name of each wine.
- `metadata` provides the metadata for the ChatGPT Retrieval Plugin document. In this case, we populate the metadata "author" field with the winery that makes each wine.

The defined fields and the values gathered from the RDF statements that match the definition are used to construct a natural language text document following a series of steps described in the *Text document assembly* section below.

For example, using the *create connector command above*, the text document for the Franvino wine data above will look like this:

```
Franvino:
- is a RedWine.
- made from grape Merlot.
- made from grape Cabernet Franc.
- has sugar dry.
- has year 2012.
```

All documents are sent to the ChatGPT Retrieval Plugin in the format expected by it. This JSON sample also illustrates how the "author" field of the metadata is populated:

```
{
  "documents" : [ {
    "metadata" : {
      "author" : "Semantinos"
    },
    "id" : "http://www.example.com/demo/wine#Yoyowine",
```

(continues on next page)

(continued from previous page)

```
    "text" : "Yoyowine:\n- is a RedWine.\n- made from grape Cabernet Sauvignon.\n- has sugar dry.\n-
↪has year 2013.\n"
  }, {
    "metadata" : {
      "author" : "Semantinos"
    },
    "id" : "http://www.example.com/demo/wine#Franvino",
    "text" : "Franvino:\n- is a RedWine.\n- made from grape Merlot.\n- made from grape Cabernet Franc.\
↪n- has sugar dry.\n- has year 2012.\n"
  }, {
    "metadata" : {
      "author" : "In vino veritas"
    },
    "id" : "http://www.example.com/demo/wine#Noirette",
    "text" : "Noirette:\n- is a RedWine.\n- made from grape Pinot Noir.\n- has sugar medium.\n- has
↪year 2012.\n"
  }, {
    "metadata" : {
      "author" : "In vino veritas"
    },
    "id" : "http://www.example.com/demo/wine#Blanquito",
    "text" : "Blanquito:\n- is a WhiteWine.\n- made from grape Chardonnay.\n- has sugar dry.\n- has
↪year 2012.\n"
  }, {
    "metadata" : {
      "author" : "In vino veritas"
    },
    "id" : "http://www.example.com/demo/wine#Rozova",
    "text" : "Rozova:\n- is a RoseWine.\n- made from grape Pinot Noir.\n- has sugar medium.\n- has year
↪2013.\n"
  } ]
}
```

Working with a secured ChatGPT Retrieval plugin

GraphDB allows the access of a secured ChatGPT Retrieval Plugin instance by passing the `retrievalBearerToken` parameter.

Instead of supplying the token as part of the connector instance configuration, you can also implement a custom authenticator class using the GraphDB Java API and set it via the `authenticationConfiguratorClass` option. See these connector authenticator examples for more information and example projects that implement such a custom class.

See the *List of creation parameters* for more information.

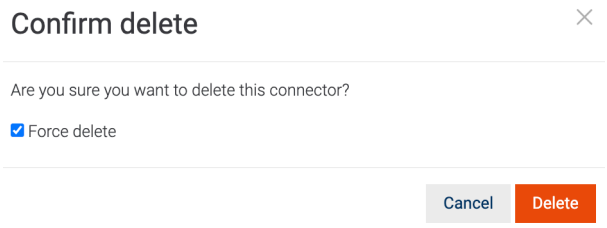
4.9.2.3 Dropping a connector instance

Dropping a connector instance removes all references to its external store from GraphDB as well as the ChatGPT Retrieval index associated with it.

The drop command is triggered by a SPARQL `INSERT` with the name of the connector instance in the subject position and `retr:dropConnector` as the predicate. For example, this removes the connector `my_index`:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:dropConnector [] .
}
```

You can also force drop a connector if a normal delete operation does not work. The force delete will remove the connector even if part of the operation fails. Go to *Setup* ▸ *Connectors* where you will see the already existing connectors that you have created. Click the *delete* icon 🗑, check *Force delete* in the dialog box, and click that dialog box's *Delete* button.



4.9.2.4 Retrieving the create options for a connector instance

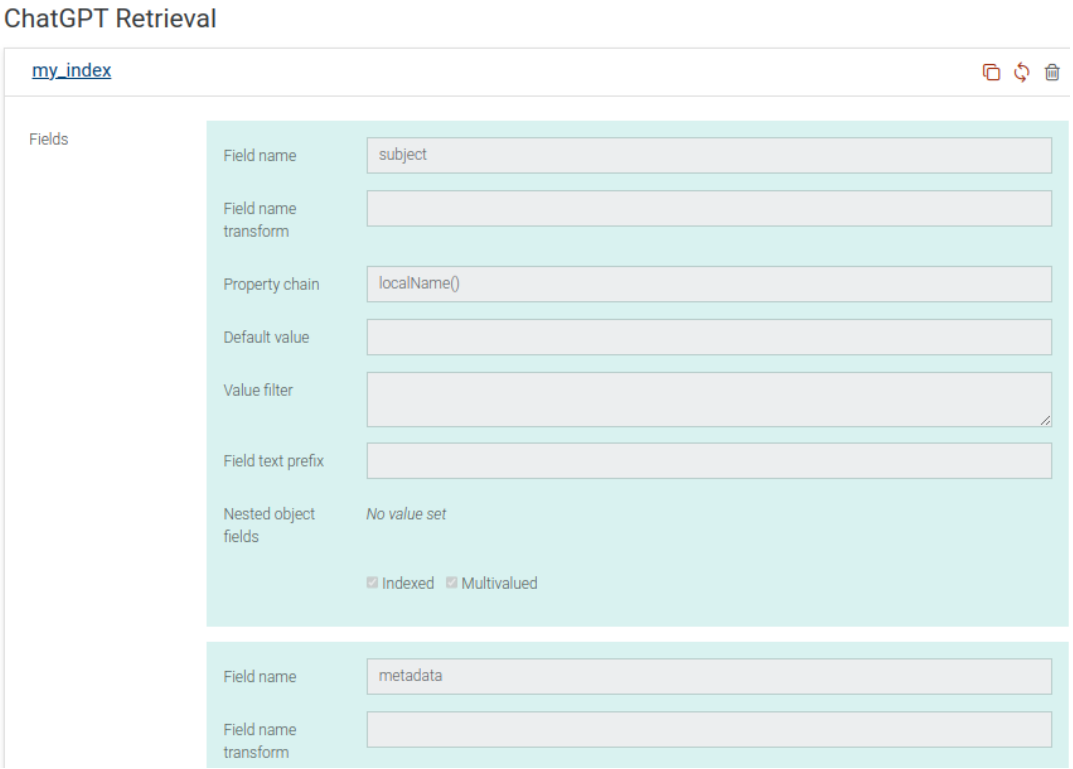
You can view the options string that was used to create a particular connector instance with the following query:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT ?createString {
  retr-index:my_index retr:listOptionValues ?createString .
}
```

4.9.2.5 Listing available connector instances

In the Connectors management view

Existing instances of a given Connector type are shown above the *New connector* button for that connector type. Click the name of an instance to view its configuration and SPARQL query or click the *Repair* 🔄 or *Delete* 🗑 icons to perform these operations. Click the *Copy* 📋 icon to copy the connector definition query to your clipboard.



With a SPARQL query

Listing connector instances returns all previously created instances. It is a `SELECT` query with the `listConnectors` predicate:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>

SELECT ?cntUri ?cntStr {
  ?cntUri retr:listConnectors ?cntStr .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance that was used during creation such as `http://www.ontotext.com/connectors/retrieval/instance#my_index`, while `?cntStr` is bound to a string representing the local name such as "my_index" in this example.

4.9.2.6 Instance status check

You can query the internal state of each connector instance using a `SELECT` query with the `connectorStatus` predicate:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>

SELECT ?cntUri ?cntStatus {
  ?cntUri retr:connectorStatus ?cntStatus .
}
```

In this example, `?cntUri` is bound to the IRI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this IRI. The status is key-value based.

4.9.3 Working with data

4.9.3.1 Adding, updating and deleting data

From the user point of view, all synchronization happens transparently without using any additional predicates or naming a specific store explicitly. You simply execute standard SPARQL `INSERT` and `DELETE` queries. This is achieved by intercepting all changes in the plugin and determining which ChatGPT Retrieval Plugin documents need to be updated.

4.9.3.2 Simple queries

Once a connector instance has been created, you can use SPARQL to query data from it. For each matching ChatGPT Retrieval Plugin document, the connector instance returns the document subject. In its simplest form, you query that data using a `SELECT` query that includes a ChatGPT Retrieval Plugin query as the object of the `retr:query` predicate:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT * {
    [] a retr-index:my_index ;
        retr:query "cabernet" ;
        retr:entities ?entity .
}
```

The result of this query binds `?entity` to the wines whose vectors are close to "cabernet". These could be `:Franvino`, `:Yoyowine` (made from Cabernet grapes) but also `:Noirette` (semantically close since it is a red wine as

well). The exact result depends on the vector model and the specific vector database used by the ChatGPT Retrieval Plugin.

The creation of such a query follows these basic steps:

1. Get a query instance of the requested connector instance by using the RDF notation `"X a Y" (= X rdf:type Y)`, where `X` is a variable and `Y` is a connector instance IRI. `X` is bound to a query instance of the connector instance.
2. Assign a query to the query instance by using the system predicate `retr:query`.
3. Request the matching entities through the `retr:entities` predicate.

You can also provide per-query search options by using one or more option predicates. The option predicates are described in detail below. You can also retrieve information about the matching chunks of text as well as the metadata of the matching documents; see *Chunk and metadata extraction* for more details.

4.9.3.3 Raw queries

To access a ChatGPT Retrieval Plugin query parameter that is not exposed through a special predicate, use a raw query. Instead of providing text in the `:query` part, specify a raw ChatGPT Retrieval Plugin query. For example, to filter by the "author" field in the metadata:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT ?entity {
  ?search a retr-index:my_index ;
        retr:query '''
            {
              "queries": [
                {
                  "query": "cabernet",
                  "filter": {
                    "author": "Semantinos"
                  }
                }
              ]
            }
        ''' ;
    retr:entities ?entity .
}
```

This query return only wines whose winery ("author" field in the metadata) is "Semantinos".

Combining ChatGPT Retrieval results with GraphDB data

You can use the bound `?entity` variable in other SPARQL triple patterns to build complex queries that join to or fetch additional data from GraphDB. For example, to see the actual grapes in the matching wines as well as the year they were made:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>
PREFIX wine: <http://www.example.com/demo/wine#>

SELECT ?entity ?grape ?year {
  ?search a retr-index:my_index ;
      retr:query "cabernet" ;
      retr:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
```

(continues on next page)

(continued from previous page)

```
  ?entity wine:hasYear ?year
}
```

The result may look like this:

	entity	grape	year
1	wine:Yoyowine	wine:CabernetSauvignon	"2013"^^xsd:integer
2	wine:Franvino	wine:Merlot	"2012"^^xsd:integer
3	wine:Franvino	wine:CabernetFranc	"2012"^^xsd:integer

Note: `:Franvino` is returned twice because it is made from two different grapes, both of which are returned.

Entity match score

You can access the match score returned by the ChatGPT Retrieval Plugin by using the `score` predicate. Higher scores mean more relevance. Because each entity has its own score, the predicate should come at the entity level. For example:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT ?entity ?score {
  ?search a retr-index:my_index ;
      retr:query "grape:cabernet" ;
      retr:entities ?entity .
  ?entity retr:score ?score
}
```

The result looks like the following; the actual scores may look different depending on the specific vector database used by the ChatGPT Retrieval Plugin:

	entity	score
1	wine:Yoyowine	"0.3979403078556061"^^xsd:double
2	wine:Franvino	"0.33036553859710693"^^xsd:double

4.9.3.4 Limit

Limit (but not offset) is supported on the ChatGPT Retrieval Plugin side of the query. This is achieved through the predicate `limit`. The following demonstrates this with a limit of `1`:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT ?entity {
  ?search a retr-index:my_index ;
      retr:query "cabernet" ;
      retr:limit 1 ;
      retr:entities ?entity .
}
```

The result contains a single wine, Franvino:

	entity
1	wine:Yoyowine
2	wine:Franvino
3	wine:Blanquito

Note: The specific order in which GraphDB returns the results depends on how the ChatGPT Retrieval Plugin returns the matches according to their match score.

4.9.3.5 Chunk and metadata extraction

The ChatGPT Retrieval Plugin can return information about the matching chunks of text as well as the metadata of the matching documents. You can access this information through the dedicated predicate `retr:snippets`. It binds a blank node that in turn provides the actual bits of information via the predicates `retr:snippetField` and `retr:snippetText`. The predicate snippets must be attached to the entity because each entity has a different set of snippets. For example, in a search for Cabernet:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a retr-index:my_index ;
      retr:query "cabernet" ;
      retr:entities ?entity .
  ?entity retr:snippets ?snippet .
  ?snippet retr:snippetField ?snippetField ;
      retr:snippetText ?snippetText .
}
```

The query returns the three wines that are semantically (vector) related to “Cabernet” as well as the respective matching chunks and metadata:

	entity	snippetField	snippetText
1	wine:Franvino	"metadata_author"	"Semantinos"
2	wine:Franvino	"chunk_id"	"http://www.ontotext.com/example/wine#Franvino_0"
3	wine:Franvino	"chunk"	"Franvino: - is a RedWine. - made from grape Merlot. - made from grape Cabernet Franc. - has sugar dry. - has year 2012."
4	wine:Yoyowine	"metadata_author"	"Semantinos"
5	wine:Yoyowine	"chunk_id"	"http://www.ontotext.com/example/wine#Yoyowine_0"
6	wine:Yoyowine	"chunk"	"Yoyowine: - is a RedWine. - made from grape Cabernet Sauvignon. - has sugar dry. - has year 2013."
7	wine:Noirette	"metadata_author"	"In vino veritas"
8	wine:Noirette	"chunk_id"	"http://www.ontotext.com/example/wine#Noirette_0"
9	wine:Noirette	"chunk"	"Noirette: - is a RedWine. - made from grape Pinot Noir. - has sugar medium. - has year 2012."

Note: The actual results might be different depending on the specific vector database used by the ChatGPT Retrieval Plugin.

4.9.4 List of creation parameters

The creation parameters define how the `retr:createConnector` predicate creates a connector instance. Some are required and some are optional. All parameters are provided together in a JSON object where the parameter names are the object keys. Parameter values may be simple JSON values such as a string or a boolean, or they can be lists or objects.

All of the creation parameters can also be set conveniently from the *Create Connector user interface* without any knowledge of JSON.

`readonly` (boolean), optional, read-only mode A read-only connector will index all existing data in the repository at creation time, but, unlike non-read-only connectors, it will:

- Not react to updates. Changes will not be synced to the connector.
- Not keep any extra structures (such as the internal data structures for tracking updates to chains).

The only way to index changes in data after the connector has been created is to repair (or drop and recreate) the connector.

`importGraph` (boolean), optional, specifies that the RDF data from which to create the connector is in a special virtual graph Used to create a connector instance from temporary RDF data inserted in the same transaction. It requires read-only mode and creates a connector whose data will come from statements inserted into a special virtual graph instead of data contained in the repository. The virtual graph is `retr:graph`, where the prefix `retr:` represents the IRI `http://www.ontotext.com/connectors/retrieval#` as before. The data must be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and the create statement must be in the same transaction. In Workbench, you can do this by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This feature requires *read-only mode*.

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
INSERT {
    GRAPH retr:graph {
        ...
    }
} WHERE {
        ...
};
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>
INSERT DATA {
    retr-index:my_index retr:createConnector '''
{
  "readonly": true,
  "importGraph": true,
  "fields": [],
  "languages": [],
  "types": [],
}
''' .
}
```

`importFile` (string), optional, an RDF file with data from which to create the connector Creates a connector whose data will come from an RDF file on the file system instead of data contained in the repository. The value must be the full path to the RDF file. This feature requires *readonly mode*.

`detectFields` (boolean), optional, detects fields This mode introduces automatic field detection when creating a connector. You can omit the specification of `fields` in JSON. Instead, you will get automatic `fields`, each corresponds to a single predicate, and its field name is the same as the predicate.

In this mode, specifying types is also optional. If types are not provided, then all types will be indexed. This mode requires *importGraph* or *importFile*.

Once the connector is created, you can inspect the detected fields in the Connector management section of Workbench.

`retrievalUrl` (string), required, the ChatGPT Retrieval Plugin instance to sync to Because the ChatGPT Retrieval Plugin is a third-party service, you must specify the URL where it is running. The node value takes the form `http://hostname.domain:port` or `https://hostname.domain:port`. No default value. Can be updated at runtime with no need to rebuild the index.

`retrievalBearerToken` (string), optional, the Bearer token to use for authentication with the ChatGPT Retrieval plugin No default value. Can be updated at runtime with no need to rebuild the index.

`bulkUpdateBatchSize` (integer), optional, controls the maximum number of documents sent per bulk request Default value is `1,000`. Can be updated at runtime with no need to rebuild the index.

`authenticationConfiguratorClass` optional, provides custom authentication behavior See *Working with a secured ChatGPT Retrieval plugin*.

`types` (list of IRIs), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of IRIs. At least one type IRI is required.

Use the pseudo-IRI `$any` to sync entities that have at least one RDF type.

Use the pseudo-IRI `$untyped` to sync entities regardless of whether they have any RDF type. See also the examples in *General full-text search with the connectors*.

`languages` (list of strings), optional, valid languages for literals RDF data is often multilingual, but only some of the languages represented in the literal values can be mapped. You can do this by specifying a list of language ranges to be matched to the language tags of literals according to RFC 4647 Section 3.3.1, Basic Filtering. You can also use an empty range to include literals that have no language tag. The list of language ranges maps all existing literals that have matching language tags.

`fields` (list of field objects), required, defines the mapping from RDF to ChatGPT Retrieval documents The fields specify exactly which parts of each entity will be synchronized as well as the specific details on the connector side. The field is the smallest synchronization unit, and it maps a property chain from GraphDB to a field in ChatGPT Retrieval. The fields are specified as a list of field objects. At least one field object is required. Each field object has further keys that specify details.

`fieldName` (string), required, the name of the field in ChatGPT Retrieval documents The name of the field that defines the mapping on the connector side. It is specified by the key `fieldName` with a string value. The field names are used to *construct text documents* so it's best to use meaningful field names that are easy to understand by a human, and hence, the GPT-4 model.

`fieldTextPrefix` (string), optional, specifies a template for *constructing the field name in text documents*, `has {}` by de
If the value contains `{}` it will be replaced by the normalized field name. Field names are normalized by converting from Java camel-case notation to separate words. For example, the field name "dateOfBirth" will be normalized to "date of birth".

`fieldNameTransform` (one of `none`, `predicate` or `predicate.localName`), optional, `none` by default
Defines an optional transformation of the field name. Although `fieldName` is always required, it is ignored if `fieldNameTransform` is `predicate` or `predicate.localName`.

- `none`: The field name is supplied via the `fieldName` option.
- `predicate`: The field name is equal to the full IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will also be `http://www.w3.org/2000/01/rdf-schema#label`.
- `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain. For example, if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`.

See *Indexing all literals in distinct fields* for an example.

`propertyChain` (list of IRI), required, defines the property chain to reach the value The property chain defines the mapping on the GraphDB side. A property chain is a sequence of triples where the entity IRI is the subject of the first triple, its object is the subject of the next triple, etc. In this model,

a property chain with a single element corresponds to a direct property defined by a single triple. Property chains are specified as a list of IRIs where at least one IRI must be provided.

The IRI of the document will be synchronized to the special field `document_id` in the ChatGPT Retrieval Plugin. You may use it to query the ChatGPT Retrieval Plugin directly and to retrieve the matching entity IRI.

See *Multiple property chains per field* to define a field whose values are populated from more than one property chain.

See *Indexing language tags* to define a field whose values are populated with the language tags of literals.

See *Indexing the IRI of an entity* to define a field whose values are populated with the IRI of the indexed entity.

See *Wildcard literal indexing* to define a field whose values are populated with literals regardless of their predicate.

- **`valueFilter` (string), optional, specifies the value filter for the field** See also *Entity filtering*.

`documentFilter` (string), optional, specifies the nested document filter for the field Only for fields that define nested documents. See also *Entity filtering*.

`defaultValue` (string), optional, specifies a default value for the field The default value (`defaultValue`) provides a means to specify a default value for the field when the property chain has no matching values in GraphDB. The default value can be a plain literal, a literal with a datatype (`xsd:` prefix supported), a literal with language, or an IRI. It has no default value.

`indexed` (boolean), optional, default `true` Indexed fields are used to construct the text document for the ChatGPT Retrieval Plugin; this is controlled by the Boolean option `indexed`. `true` by default. Non-indexed fields can be used to perform filtering without affecting the document contents.

Setting this to `false` on the special `metadata` field or any of its nested fields will be ignored.

`multivalued` (boolean), optional, default `true` RDF properties and synchronized fields may have more than one value. If `multivalued` is set to `true`, all values will be used for the text document. If set to `false`, only a single value will be synchronized. `true` by default.

`objectFields` (objects array), optional, nested object mapping Provide a mapping for the nested object's fields. At present, nested objects can be defined only for the metadata of the ChatGPT Retrieval plugin.

`valueFilter` (string), optional, specifies the top-level value filter for the document See also *Entity filtering*.

`documentFilter` (string), optional, specifies the top-level document filter for the document See also *Entity filtering*.

4.9.4.1 Updating parameters at runtime

As mentioned above, you can update the following connection parameters at runtime with no need to rebuild the index:

- `retrievalUrl`
- `retrievalBearerToken`
- `bulkUpdateBatchSize`

The following SPARQL update request demonstrates this by updating the Bearer token:

```
PREFIX conn:<http://www.ontotext.com/connectors/retrieval#>
PREFIX inst:<http://www.ontotext.com/connectors/retrieval/instance#>
INSERT DATA {
inst:proper_index conn:updateConnector '''
    {
```

(continues on next page)

(continued from previous page)

```
        "retrievalBearerToken": "<my-token>"
    }
''' .
}
```

4.9.4.2 Special field definitions

Helper field mappings

The ChatGPT Retrieval connector reserves some field names for fields that are treated differently when *constructing a text document*.

- `subject`: the value of this field will be at the top of the text document followed by a colon without mentioning the field name.
- `text`: the value of this field will be appended at the end of the document without mentioning the field name.
- `metadata`: the value of this field will be used to construct the metadata of the document and will not be used to generate the resulting text. Note that the nested fields that are contained in it must follow the ChatGPT Retrieval Plugin metadata schema. The value for the `created_at` field of the metadata schema must be an `xsd:dateTime`. If the literal used to construct it is a valid `xsd:date`, it will be padded to make it a `xsd:dateTime` value by appending "T00:00:00Z" to conform to the requirement.

Nested objects

Nested objects are connector documents that are used as values in the main document. They are defined with the *objectFields* option.

In the example under *Using the create command* earlier, `objectFields` is used to map `author` as a nested object within the `metadata` field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in ChatGPT Retrieval) with more than one property chain — for example, the concept of "name" could be defined as a single canonical name, multiple historical names and some unofficial names. If you want to index these together as a single field in ChatGPT Retrieval, you can define this as a multiple property chains field.

Fields with multiple property chains are defined as a set of separate *virtual* fields that will be merged into a single *physical* field when indexed. Virtual fields are distinguished by the suffix `$xyz`, where `xyz` is any convenient alphanumeric sequence. For example, we can define the fields `name$1` and `name$2` like this:

```
...
  "fields": [
    {
      "fieldName": "name$1",
      "propertyChain": [
        "http://www.example.com/demo#canonicalName"
      ],
      "fieldName": "name$2",
      "propertyChain": [
        "http://www.example.com/demo#historicalName"
      ]
      ...
    },
...
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the `name` field in ChatGPT Retrieval.

Note: You cannot mix suffixed and unsuffixed fields with the same name. For example, if you defined `myField$new` and `myField$old`, you cannot have a field called just `myField`.

Filters and fields with multiple property chains

You can use filters with fields defined with multiple property chains. Both the physical field values and the individual virtual field values are available:

- Physical fields are specified without the suffix (for example, `?myField`)
- Virtual fields are specified with the suffix (for example, `?myField$2` or `?myField$alt`)

Note: Physical fields cannot be combined with `parent()` as their values come from different property chains. If you really need to filter the same parent level, you can rewrite `parent(?myField) in (<urn:x>, <urn:y>)` as `parent(?myField$1) in (<urn:x>, <urn:y>) || parent(?myField$2) in (<urn:x>, <urn:y>) || parent(?myField$3) ...` and surround it with parentheses if it is a part of a bigger expression.

Indexing language tags

The language tag of an RDF literal can be indexed by specifying a property chain whose last element is the pseudo-IRI `lang()`. The property preceding `lang()` must lead to a literal value. For example:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:createConnector '''
    {
      "retrievalUrl": "http://localhost:8000",
      "retrievalBearerToken": "<replace-this-with-actual-value>",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameLanguage",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "lang()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the language tag of each literal value of the property `http://www.example.com/demo#name` into the field `nameLanguage`.

Indexing named graphs

You can index a given value's named graph by ending a property chain with the special pseudo-URI `graph()`. Indexing the named graph of the value instead of the value itself allows searching by named graph.

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:createConnector '''
    {
      "retrievalUrl": "http://localhost:8000",
      "retrievalBearerToken": "<replace-this-with-actual-value>",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "nameGraph",
           "propertyChain": [
             "http://www.example.com/demo#name",
             "graph()"
           ]
         }
      ],
    }
  ''' .
}
```

The above connector will index the named graph of each value of the property `http://www.example.com/demo#name` into the field `nameGraph`.

Indexing local names

You can index the local name of a given IRI value by ending a property chain with the special pseudo-URI `localName()`. Indexing the local name instead of the full IRI is convenient when the local name is a human-readable meaningful string.

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:createConnector '''
    {
      "retrievalUrl": "http://localhost:8000",
      "retrievalBearerToken": "<replace-this-with-actual-value>",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
         {
           "fieldName": "name",
           "propertyChain": [
             "http://www.example.com/demo#name"
           ]
         },
         {
           "fieldName": "feature",
```

(continues on next page)

(continued from previous page)

```
          "propertyChain": [
            "http://www.example.com/demo#feature",
            "localName()"
          ]
        }
      ],
    }
  ''' .
}
```

The above connector will index the local name of each IRI value of the property `http://www.example.com/demo#feature` into the field `feature`.

Wildcard literal indexing

In this mode, the last element of a property chain is a wildcard that will match any predicate that leads to a literal value. Use the special pseudo-IRI `$literal` as the last element of the property chain to activate it.

Note: Currently, it really means any literal, including literals with data types.

For example:

```
{
    "fields" : [ {
        "propertyChain" : [ "$literal" ],
        "fieldName" : "name"
    }, {
        "propertyChain" : [ "http://example.com/description", "$literal" ],
        "fieldName" : "description"
    }
    ...
}
```

See *Indexing all literals* for a detailed example.

Indexing the IRI of an entity

Sometimes you may need the IRI of each entity (for example, `http://www.example.com/demo/wine#Franvino` from our small example dataset) indexed as a regular field. Do this by specifying a property chain with a single property referring to the pseudo-IRI `$self`. For example:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
    retr-index:my_index retr:createConnector '''
{
  "retrievalUrl": "http://localhost:8000",
  "retrievalBearerToken": "<replace-this-with-actual-value>",
  "types": [
    "http://www.example.com/demo/wine#Wine"
  ],
  "fields": [
    {
      "fieldName": "entityId",
      "propertyChain": [
```

(continues on next page)

(continued from previous page)

```
        "$self"
      ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.example.com/demo/wine#madeFromGrape",
        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    },
  ]
}
''' .
}
```

The above connector will index the IRI of each wine into the field `entityId`.

Note: Note that GraphDB will also use the IRI of each entity as the ID of each document in ChatGPT Retrieval, which is represented by the field `id`.

4.9.5 Entity filtering

The ChatGPT Retrieval connector supports four kinds of entity filters used to fine-tune the set of entities and individual values for the configured fields, based on the field value. Entities and field values are synchronized to ChatGPT Retrieval if, and only if, they pass the filter. The filters are similar to a SPARQL query's `FILTER()` but not exactly the same. You can reference each configured field in them by prefixing its name with a `?`, much like referencing a SPARQL variable. Entity filter examples are provided at the end of this section.

4.9.5.1 Types of filters

Top-level value filter The top-level value filter is specified with the `valueFilter` parameter. It is evaluated prior to anything else when only the document ID is known, and it may not refer to any field names but only to the special field `$this` that contains the current document ID. Failing to pass this filter removes the entire document early in the indexing process. It can be used to introduce more restrictions similar to the built-in filtering by type via the `types` property.

Top-level document filter The top-level document filter is specified with the `documentFilter` parameter. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. You can use it to enforce global document restrictions such as that certain fields are required or a document needs to be indexed only if a certain field value meets specific conditions.

Per-field value filter The per-field value filter is specified with the `valueFilter` parameter inside the field definition of the field whose values are to be filtered. The filter is evaluated while collecting the data for the field when each field value becomes available.

The variable that contains the field value is `$this`. You can use other field names to filter the current field's value based on the value of another field. For example, `$this > ?age` will compare the current field value to the value of the field age (see also *Two-variable filtering*). Failing to pass the filter will remove the current field value.

On nested documents, the per-field value filter can be used to remove the entire nested document early in the indexing process, e.g., by checking the type of the nested document via next hop with `rdf:type`.

Nested document filter The nested document filter is specified with the `documentFilter` parameter **inside the field definition** of the field that defines the root of a nested document. The filter is evaluated after the entire nested document has been collected. Failing to pass this filter removes the entire nested document.

Inside a nested document filter, the field names are within the context of the nested document and not within the context of the top-level document. For example, if you have a `children` field that defines a nested document and you use a filter like `?age < "10"^^xsd:int`, it will refer to the field `children.age`. You can use the prefix `$outer.` one or more times to refer to field values from the outer document from the viewpoint of the nested document. For example, `$outer.age > "25"^^xsd:int` will refer to the `age` field that is a sibling of the `children` field.

Other than the above differences, the nested document filter is equivalent to the top-level document filter from the viewpoint of the nested document.

4.9.5.2 Filter operators

The filter operators test whether a given field's value satisfies a certain condition.

Field comparisons are done on the original RDF values before they are converted to textual representation.

Operator	Meaning
`?var in (value1, value2, ...)`	Tests if the `var` field's value is one of the specified values. Unlike the similar SPARQL operator, values are compared strictly — like the SPARQL `sameTerm()` function, values that do not match are treated as if they were not present in the repository. Example: `?status in ("active", "new")`
`?var not in (value1, value2, ...)`	The negated version of the `in` operator. Example: `?status not in ("archived")`
`bound(?var)`	Tests if the field `var` has a valid value. This can be used to make the field compulsory. Example: `bound(?name)`
`isExplicit(?var)`	Tests if the `var` field's value came from an explicit statement. This will use the last element of the property chain. If you need to assert the explicit status of a previous property chain use `parent(?var)` as many times as needed. Example: `isExplicit(?name)`

Continued on next page

Table 15 – continued from previous page

Operator	Meaning

`?var = value` (equal to)
`?var != value` (not equal to)
`?var > value` (greater than)
`?var >= value` (greater than or equal to)
`?var < value` (less than)
`?var <= value` (less than or equal to)
	RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The `var` field's value will be compared to the specified RDF value. When comparing RDF values that are literals, their datatypes must be compatible, e.g., `xsd:integer` and `xsd:long` but not `xsd:string` and `xsd:date`. Values that do not match are treated as if they were not present in the repository.
	
	Examples:
	Given that `height` has a value of `"150"^^xsd:int` and `dateOfBirth` has a
	value of `"1989-12-31"^^xsd:date`, then:
	
	`?height = "150"^^xsd:int` is `true`
	`?height = "150"^^xsd:long` is `true`
	`?height = "150"` is `false`
	
	`?height != "151"^^xsd:int` is `true`
	`?height != "150"` is `true`
	
	`?height > "150"^^xsd:int` is `false`
	`?height >= "150"^^xsd:int` is `true`
	`?dateOfBirth < "1990-01-01"^^xsd:date` is `true`

`regex(?var, "pattern")`
or
`regex(?var, "pattern", "i")`
	Tests if the `var` field's value matches the given regular expression pattern.
	The "i" flag tells the function to ignore case when matching. Values that do
	not match are treated as if they were not present in the repository.
	
	Example:
	`regex(?name, "^mrs?", "i")`

`expr1 || expr2`
or
`expr1 or expr2`
	Logical disjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?name) || bound(?company)`
	`bound(?name) or bound(?company)`

`expr1 && expr2`
or
`expr1 and expr2`
	Logical conjunction of expressions `expr1` and `expr2`.
	
	Examples:
	`bound(?status) && ?status in ("active", "new")`
	`bound(?status) and ?status in ("active", "new")`

`!expr`	Logical negation of expression `expr`. Example: `!bound(?company)`

Continued on next page

Table 15 – continued from previous page

Operator	Meaning
`( expr )`	Grouping of expressions Example: `(bound(?name) or bound(?company)) && bound(?address)`

4.9.5.3 Filter modifiers

Along with the filter operators, additional constructions let you write filters based not on the values of a field but on values related to them.

Accessing the previous element in the chain The construction `parent(?var)` goes to a previous level in a property chain. You can apply it recursively as many times as as needed; for example, `parent(parent(parent(?var)))` goes back in the chain three times. You can use the effective value of `parent(?var)` with the `in` or `not in` operators (for example, `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator (for example, `parent(bound(?var))`).

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) accesses additional values through the `uri` property. This construction roughly corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the `var` field. You can use the effective value of `?var -> uri` with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. You can combine it with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. You can apply the same construction to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or you can combine it with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (or just `type`), which will be expanded to `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`.

Filtering by RDF graph The construction `graph(?var)` gives you access to the RDF graph of a field's value. A typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)` but using `isExplicit(?a)` is the recommended way.

You can combine this construction with the `parent()` function like this: `graph(parent(?a)) in (<urn:a>)`.

Filtering by language tags The construction `lang(?var)` gives you access to the language tag of field's value. (Only RDF literals can have a language tag.) The typical use case is to sync only values written in a given language: `lang(?a) in ("de", "it", "no")`. You can combine this construction with the `parent()` function and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. You can filter literal values without language tags by using an empty tag: `""`.

Current context variable `$this` You can use the special field variable `$this` (and not `?this`, `?$this`, or `$?this`) to refer to the current context. In the top-level value filter and the top-level document filter, it refers to the document. In the per-field value filter, it refers to the currently filtered field value. In the nested document filter, it refers to the nested document.

`ALL()` quantifier In the context of document-level filtering, a match is `true` if at least one of potentially many field values match. For example, `?location = <urn:Europe>` would return `true` if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

You can also use the `ALL()` quantifier when you need all values to match. For example, `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values You can combine entity filters with default values for more flexible behavior.

If a field has no values in the RDF database, the `defaultValue` is used. But if a field has some values, `defaultValue` is NOT used, even if all values are filtered out. See an example in *Basic entity filter*.

A typical use case for an entity filter is having soft deletes — instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

4.9.5.4 Two-variable filtering

Besides comparing a field value to one or more constants, or running an existential check on the field value, some use cases also require the ability to compare the field value with the value of another field in order to produce the desired result. GraphDB solves this by supporting two-variable filtering in the per-field value filter, the top-level document filter, and the nested document filter.

Note: This type of filtering is not possible in the top-level value filter because the only variable that is available there is `$this`.

In the top-level document filter and the nested document filter, there are no restrictions because all values are available at the time of evaluation.

In the per-field value filter, two-variable filtering will reorder the defined fields such that values for other fields are already available when the current field's filter is evaluated. For example, let's say we defined a filter `$this > ?salary` for the field `price`. This will force the connector to process the field `salary` first, apply its per-field value filter if any, and only then start collecting and filtering the values for the field `price`.

Cyclic dependencies will be detected and reported as an invalid filter. For example, if in addition to the above we define a per-field value filter `?price > "1000"^^xsd:int` for the field `salary`, a cyclic dependency will be detected as both `price` and `salary` will require the other field to be indexed first.

4.9.5.5 Basic entity filter example

Let's look at an example that uses the following RDF data.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example: <http://www.example.com/demo#> .

# the entity below will be synchronised because it has a matching value for city: ?city in ("London")
example:alpha
    rdf:type example:gadget ;
    example:name "John Synced" ;
    example:city "London" .

# the entity below will not be synchronised because it lacks the property completely: bound(?city)
example:beta
    rdf:type example:gadget ;
    example:name "Peter Syncfree" .

# the entity below will not be synchronized because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
example:gamma
    rdf:type example:gadget ;
    example:name "Mary Syncless" ;
    example:city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:createConnector '''
    {
      "retrievalUrl": "http://localhost:8000",
      "retrievalBearerToken": "<replace-this-with-actual-value>",
      "types": ["http://www.example.com/demo#gadget"],
      "fields": [
```

(continues on next page)

(continued from previous page)

```
        {
          "fieldName": "name",
          "propertyChain": ["http://www.example.com/demo#name"]
        },
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo#city"],
          "valueFilter": "$this = \\"London\\""
        }
      ],
      "documentFilter": "bound(?city)"
    }
  ''' .
}
```

The `:beta` entity is not synchronized because it has no value for `city`.

To handle such cases, you can modify the connector configuration to specify a default value for `city`:

```
...
        {
          "fieldName": "city",
          "propertyChain": ["http://www.example.com/demo#city"],
          "defaultValue": "London"
        }
...
}
```

The default value is used for the entity `:beta` as it has no value for city in the repository. As the value is "London", the entity is synchronized.

4.9.5.6 Advanced entity filter example

Sometimes, data represented in RDF does not map directly to a non-RDF format. For example, if you have news articles and they can be tagged with different concepts such as locations, persons, and events, one possible way to model this is a single property `:taggedWith`. Consider the following RDF data:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix example2: <http://www.example.com/demo2#> .

example2:Berlin
    rdf:type example2:Location ;
    rdfs:label "Berlin" .

example2:Mozart
    rdf:type example2:Person ;
    rdfs:label "Wolfgang Amadeus Mozart" .

example2:Einstein
    rdf:type example2:Person ;
    rdfs:label "Albert Einstein" .

example2:Cannes-FF
    rdf:type example2:Event ;
    rdfs:label "Cannes Film Festival" .

example2:Article1
    rdf:type example2:Article ;
    rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
```

(continues on next page)

(continued from previous page)

```
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Einstein ;
    example2:taggedWith example2:Cannes-FF .

example2:Article2
    rdf:type example2:Article ;
    rdfs:comment "An article about Berlin." ;
    example2:taggedWith example2:Berlin .

example2:Article3
    rdf:type example2:Article ;
    rdfs:comment "An article about Mozart's life." ;
    example2:taggedWith example2:Mozart .

example2:Article4
    rdf:type example2:Article ;
    rdfs:comment "An article about classical music in Berlin." ;
    example2:taggedWith example2:Berlin ;
    example2:taggedWith example2:Mozart .

example2:Article5
    rdf:type example2:Article ;
    rdfs:comment "A boring article that has no tags." .

example2:Article6
    rdf:type example2:Article ;
    rdfs:comment "An article about the Cannes Film Festival in 2013." ;
    example2:taggedWith example2:Cannes-FF .
```

Assume you want to map this data to the ChatGPT Retrieval Plugin, so that the property `example2:taggedWith x` is mapped to separate fields `taggedWithPerson` and `taggedWithLocation`, according to the type of `x` (where we are not interested in Events). You can map `taggedWith` twice to different fields and then use an entity filter to get the desired values:

```
PREFIX retr: <http://www.ontotext.com/connectors/retrieval#>
PREFIX retr-index: <http://www.ontotext.com/connectors/retrieval/instance#>

INSERT DATA {
  retr-index:my_index retr:createConnector '''
    {
      "retrievalUrl": "http://localhost:8000",
      "retrievalBearerToken": "<replace-this-with-actual-value>",
      "types": ["http://www.example.com/demo2#Article"],
      "fields": [
         {
            "fieldName": "comment",
            "propertyChain": ["http://www.w3.org/2000/01/rdf-schema#comment"]
         },
         {
           "fieldName": "taggedWithPerson",
           "propertyChain": ["http://www.example.com/demo2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo2#Person>"
         },
         {
           "fieldName": "taggedWithLocation",
           "propertyChain": ["http://www.example.com/demo2#taggedWith"],
           "valueFilter": "$this -> type = <http://www.example.com/demo2#Location>"
         }
      ]
    }
  ''' .
```

(continues on next page)

(continued from previous page)

```
}
```

Note: `type` is the short way to write `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`.

The six articles in the RDF data above will be mapped as such:

Article IRI	Value in taggedWithPerson	Value in taggedWithLocation	Explanation
`:Article1`	`:Einstein`	`:Berlin`	`:taggedWith` has the values `:Einstein`, `:Berlin` and `:Cannes-FF`. The filter leaves only the correct values in the respective fields. The value `:Cannes-FF` is ignored because it does not match the filter.
`:Article2`		`:Berlin`	`:taggedWith` has the value `:Berlin`. After the filter is applied, only `taggedWithLocation` is populated.
`:Article3`	`:Mozart`		`:taggedWith` has the value `:Mozart`. After the filter is applied, only `taggedWithPerson` is populated.
`:Article4`	`:Mozart`	`:Berlin`	`:taggedWith` has the values `:Berlin` and `:Mozart`. The filter leaves only the correct values in the respective fields.
`:Article5`			`:taggedWith` has no values. The filter is not relevant.
`:Article6`			`:taggedWith` has the value `:Cannes-FF`. The filter removes it because it does not match.

4.9.6 Text document assembly

The natural language text document to pass to ChatGPT is assembled from the defined connector fields with the following steps:

1. The `subject` field value, followed by a colon and a new line, starts a series of statements.
2. Regular fields (not *Helper field mappings*) are appended next with their values in the format:

   `- has <normalized field name> <value>.`

   Nested fields are represented as `- has <normalized field name>:` followed by each field within the nested field on a new line with additional indentation.

   The `has <normalized field name>` part is configurable via the connector's *fieldTextPrefix* option.
3. Finally, for each value of the field "text", the value on its own line.

A generic text document will look like this:

```
<subject-field-value>:
- has <field-name1> <value>.
- has <field-name1> <value>.
- has <field-name2> <value>.
- has <nested-field-name1>:
    - has <inner-field-name1> <value>;
    - has <inner-field-name2> <value>.
...

<text-field-value1>
```

(continues on next page)

(continued from previous page)

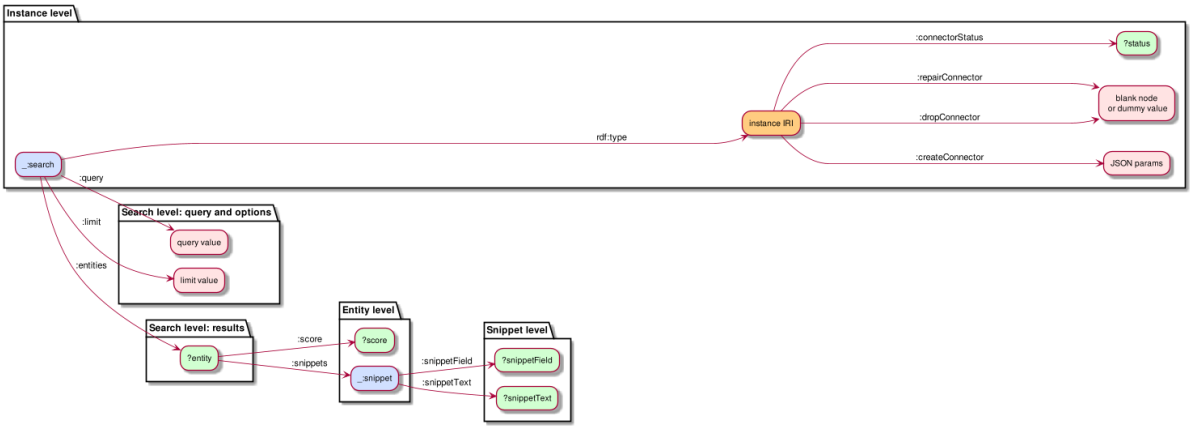
```
<text-field-value2>

...
```

See the *wines connector example* for a text document constructed from actual data.

4.9.7 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (create, drop, check the status of) connector instances or issue queries and retrieve results. It can be used as a quick reference of what a particular predicate needs to be attached to. For example, to retrieve entities, you need to use `:entities` on a search instance, and to retrieve snippets (chunks and metadata), you need to use `:snippets` on an entity.

Variables that are bound as a result of a query are shown in green, blank helper nodes are shown in blue, literals in red, and IRIs in orange. The predicates are represented by labeled arrows.



4.9.8 Caveats

4.9.8.1 Order of control

Even though SPARQL per se is not sensitive to the order of triple patterns, the ChatGPT Retrieval GraphDB Connector expects to receive certain predicates before others so that queries can be executed properly. In particular, predicates that specify the query or query options must come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.

4.10 MongoDB integration

The MongoDB integration feature is a GraphDB plugin that lets you use SPARQL to query MongoDB databases with heterogeneous joins of that data to RDF data. This section describes how to configure GraphDB and MongoDB to work together.

MongoDB is a document-based database with a huge community of developers and users. It is part of the MEAN technology stack and guarantees scalability and performance well beyond the throughput supported in GraphDB.

MongoDB is a NoSQL JSON document store and does not natively support joins, SPARQL, or RDF-enabled linked data. The integration between GraphDB and MongoDB is done by a plugin that sends a request to MongoDB and then transforms the result to an RDF model.

4.10.1 Usage

The basic steps for using MongoDB with GraphDB are:

1. Installing MongoDB
2. Preparing and loading JSON-LD documents in MongoDB
3. Configuring GraphDB with MongoDB connection settings by creating an index

In order to be converted to RDF models, the documents in MongoDB should be in the JSON-LD format.

JSON-LD documents have a hierarchical structure, which allows more complex querying of embedded and nested documents.

Each document can be in a separate context. That way, the relation between statements in GraphDB and documents in MongoDB is preserved when extracting parts of the documents and importing them into GraphDB in order to make inferred statements. The import of parts is an option for future development.

Below we see a sample document in MongoDB from the LDBC SPB benchmark:

```
{
        "_id": { "$oid": "5c0fb7f329298f15dc37bb81"},
        "@graph":
        [{
                "@id": "http://www.bbc.co.uk/things/1#id",
                "@type": "cwork:NewsItem",
                "bbc:primaryContentOf":
                [{
                        "@id": "bbcd:3#id",
                        "bbc:webDocumentType": {
                                "@id": "bbc:HighWeb"
                        }
                },
                {
                        "@id": "bbcd:4#id",
                        "bbc:webDocumentType": {
                                "@id": "bbc:Mobile"
                        }
                }],
                "cwork:about":
                [{
                        "@id": "dbpedia:AccessAir"
                },
                {
                        "@id": "dbpedia:Battle_of_Bristoe_Station"
                },
                {
                        "@id": "dbpedia:Nicolas_Bricaire_de_la_Dixmerie"
                },
                {
                        "@id": "dbpedia:Bernard_Roberts"
                },
                {
                        "@id": "dbpedia:Bartolomé_de_Medina"
                },
                {
                        "@id": "dbpedia:Don_Bonker"
                },
                {
                        "@id": "dbpedia:Cornel_Nistorescu"
                },
                {
                        "@id": "dbpedia:Clete_Roberts"
                },
```

(continues on next page)

(continued from previous page)


```
                {
                        "@id": "dbpedia:Mark_Palansky"
                },
                {
                        "@id": "dbpedia:Paul_Green_(taekwondo)"
                },
                {
                        "@id": "dbpedia:Mostafa_Abdel_Satar"
                },
                {
                        "@id": "dbpedia:Tommy_O'Connell_(hurler)"
                },
                {
                        "@id": "dbpedia:Ahmed_Ali_Salaad"
                }],
                "cwork:altText": "thumbnail atlText for CW http://www.bbc.co.uk/context/1#id",
                "cwork:audience": {
                        "@id": "cwork:NationalAudience"
                },
                "cwork:category": {
                        "@id": "http://www.bbc.co.uk/category/Company"
                },
                "cwork:dateCreated": {
                        "@type": "xsd:dateTime",
                        "@value": "2011-02-15T07:13:29.495+02:00"
                },
                "cwork:dateModified": {
                        "@type": "xsd:dateTime",
                        "@value": "2012-02-14T12:43:13.165+02:00"
                },
                "cwork:description": " constipate meant breaking felt glitzier democrat's huskily
↪breeding solicit gargling.",
                "cwork:liveCoverage": {
                        "@type": "xsd:boolean",
                        "@value": "false"
                },
                "cwork:mentions": {
                        "@id": "geonames:2862704/"
                },
                "cwork:primaryFormat":
                [{
                        "@id": "cwork:TextualFormat"
                },
                {
                        "@id": "cwork:InteractiveFormat"
                }],
                "cwork:shortTitle": " closest subsystem merit rebuking disengagement cerebrums caravans
↪conduction disbelieved might.",
                "cwork:thumbnail": {
                        "@id": "bbct:1361611547"
                },
                "cwork:title": "Beckhoff greatly agitators constructed racquets industry restrain spews
↪pitifully undertone stultification."
        }],
        "@id": "bbcc:1#id",
        "@context": {
                "bbcevent": "http://www.bbc.co.uk/ontologies/event/",
                "geo-pos": "http://www.w3.org/2003/01/geo/wgs84_pos#",
                "bbc": "http://www.bbc.co.uk/ontologies/bbc/",
                "time": "http://www.w3.org/2006/time#",
                "event": "http://purl.org/NET/c4dm/event.owl#",
```


(continues on next page)

(continued from previous page)

```
                "music-ont": "http://purl.org/ontology/mo/",
                "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
                "foaf": "http://xmlns.com/foaf/0.1/",
                "provenance": "http://www.bbc.co.uk/ontologies/provenance/",
                "owl": "http://www.w3.org/2002/07/owl#",
                "cms": "http://www.bbc.co.uk/ontologies/cms/",
                "news": "http://www.bbc.co.uk/ontologies/news/",
                "cnews": "http://www.bbc.co.uk/ontologies/news/cnews/",
                "cconcepts": "http://www.bbc.co.uk/ontologies/coreconcepts/",
                "dbp-prop": "http://dbpedia.org/property/",
                "geonames": "http://sws.geonames.org/",
                "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
                "domain": "http://www.bbc.co.uk/ontologies/domain/",
                "dbpedia": "http://dbpedia.org/resource/",
                "geo-ont": "http://www.geonames.org/ontology#",
                "bbc-pont": "http://purl.org/ontology/po/",
                "tagging": "http://www.bbc.co.uk/ontologies/tagging/",
                "sport": "http://www.bbc.co.uk/ontologies/sport/",
                "skosCore": "http://www.w3.org/2004/02/skos/core#",
                "dbp-ont": "http://dbpedia.org/ontology/",
                "xsd": "http://www.w3.org/2001/XMLSchema#",
                "core": "http://www.bbc.co.uk/ontologies/coreconcepts/",
                "curric": "http://www.bbc.co.uk/ontologies/curriculum/",
                "skos": "http://www.w3.org/2004/02/skos/core#",
                "cwork": "http://www.bbc.co.uk/ontologies/creativework/",
                "fb": "http://rdf.freebase.com/ns/",
                "ot": "http://www.ontotext.com/",
                "ldbcspb": "http://www.ldbcouncil.org/spb#",
                "bbcd": "http://www.bbc.co.uk/document/",
                "bbcc": "http://www.bbc.co.uk/context/",
                "bbct": "http://www.bbc.co.uk/thumbnail/"
        }
}
```

Note the following about the sample MongoDB document above:

- `_id` key is a MongoDB internal key.
- `@graph` node represents the RDF context in the JSON-LD document.
- `@type xsd:dateTime` date has a `@date` key with an `ISODate(...)` value. This is not related to the JSON-LD standard and is ignored when the document is parsed as an RDF model. The dates are extended for faster search and sorting. The `ISODate` in MongoDB is its internal way to store dates and is optimized for searching. This step will make querying and sorting by this date field easier but is optional.

Note: The keys in MongoDB cannot contain `"."` nor start with `"$"`. Although the JSON-LD standard allows it, MongoDB does not. Therefore, either use namespaces (see the sample above) or encoding the `.` and `$`, respectively. Only the JSON keys are subject to decoding.

4.10.2 Setup and maintenance

4.10.2.1 Installing MongoDB

The instructions below assume that you have the MongoDB server installed and running on a computer you can access. Setting up and maintaining a MongoDB database is a separate task and must be accomplished outside of GraphDB. See the MongoDB documentation for details.

The GraphDB integration plugin uses MongoDB Java driver version 3.8. More information about the compatibility between MongoDB Java driver and MongoDB version is available in the MongoDB compatibility tables.

4.10.2.2 Creating an index

To configure GraphDB with MongoDB connection settings, you need to set:

- The server where MongoDB is running
- The port on which MongoDB is listening
- The name of the database you are using
- The name of the MongoDB collection you are using
- The credentials (optional unless you are using authentication) — the username and password that will allow you to connect to the database

This sample update request demonstrates how to create a MongoDB index:

```
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
    mongodb-index:spb1000 mongodb:service "mongodb://localhost:27017" ;
        mongodb:database "ldbc" ;
        mongodb:collection "creativeWorks" .
}
```

Supported predicates:

- `:service` — MongoDB connection string
- `:database` — MongoDB database
- `:collection` — MongoDB collection
- `:user` — (optional) MongoDB user for the connection
- `:password` — (optional) the user's password
- `:authDb` — (optional) the database where the user is authenticated

4.10.2.3 Upgrading an index

When upgrading to a newer GraphDB version, it may contain plugins that are not present in the older version. If so, the PluginManager disables the newly detected plugin, so you need to enable it by executing the following SPARQL query:

```
insert data { [] <http://www.ontotext.com/owlim/system#startplugin> "mongodb" }
```

Then create the plugin by executing the SPARQL update request above under *Creating an index*. Also, make sure that you don't delete the database in the plugin you are using.

4.10.2.4 Deleting an index

Delete a MongoDB index using the `mongodb:drop` predicate in an update request like the following:

```
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
    mongodb-index:spb1000 mongodb:drop [] .
}
```

4.10.2.5 Loading sample data

MongoDB's mongoimport tool lets you import data into collections from the command line. The following command will import the `cwork1000.json` file, which has 1,000 CreativeWork documents, in MongoDB database "ldbc" and the collection "creativeWorks".

```
mongoimport --db ldbc --collection creativeWorks --file cwork1000.json
```

4.10.2.6 Querying MongoDB

This sample query returns the `dateModified` for documents with the specified audience:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>

SELECT ?creativeWork ?modified WHERE {
        ?search a mongodb-index:spb1000 ;
                mongodb:find '{"@graph.cwork:audience.@id" : "cwork:NationalAudience"}' ;
                mongodb:entity ?entity .
        GRAPH mongodb-index:spb1000 {
                ?creativeWork cwork:dateModified ?modified .
        }
}
```

	creativeWork	modified
1	http://www.bbc.co.uk/things/7#id	"2011-05-24T20:58:30.144+03:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
3	http://www.bbc.co.uk/things/14#id	"2011-12-27T20:21:48.733+02:00"^^xsd:dateTime
4	http://www.bbc.co.uk/things/11#id	"2012-01-23T10:16:23.268+02:00"^^xsd:dateTime
5	http://www.bbc.co.uk/things/15#id	"2011-04-02T09:47:34.612+03:00"^^xsd:dateTime
6	http://www.bbc.co.uk/things/25#id	"2012-01-04T13:52:07.957+02:00"^^xsd:dateTime
7	http://www.bbc.co.uk/things/29#id	"2012-02-04T16:13:33.701+02:00"^^xsd:dateTime
8	http://www.bbc.co.uk/things/31#id	"2011-12-27T09:36:20.211+02:00"^^xsd:dateTime
9	http://www.bbc.co.uk/things/30#id	"2011-04-23T08:12:33.724+03:00"^^xsd:dateTime
10	http://www.bbc.co.uk/things/34#id	"2012-02-28T02:53:15.117+02:00"^^xsd:dateTime

In a query, use the exact values from the documents. For example, if the full URIs are used instead of "`cwork:NationalAudience`" or "`@graph.cwork:audience.@id`", there would not be any matching results.

The `:find` argument is a valid JSON document.

Note: The results are returned in a named graph to indicate when the plugin should bind the variables. This is an API plugin limitation. The variables to be bound by the plugin are in a named graph. This allows GraphDB to determine whether to bind the specific variable using MongoDB or not.

Supported predicates:

- `mongodb:find`: Accepts a single JSON object provided as an RDF literal and sets a query string. The value is used to call `db.collection.find()`.
- `mongodb:project`: Accepts a single JSON object provided as an RDF literal. The value is used to select the projection for the results returned by `mongodb:find`. Find more information at MongoDB: Project Fields to Return from Query.
- `mongodb:aggregate`: Accepts an array of JSON objects. Calls `db.collection.aggregate()`. This is the most flexible way to make a MongoDB query, because the `find()` method is just a single phase of the aggregation pipeline. The `mongodb:aggregate` predicate takes precedence over `mongodb:find` and `mongodb:project`. This means that if both `mongodb:aggregate` and `mongodb:find` are used, `mongodb:find` will be ignored.
- `mongodb:graph`: Accepts an IRI. Specifies the IRI of the named graph in which the bound variables should be. Its default value is the name of the index itself.
- `mongodb:entity` : Returns the IRI of the MongoDB document. If the JSON-LD has context, the value of `@graph.@id` is used. In case of multiple values, the first one is chosen and a warning is logged. If the JSON-LD has no context, the value of the `@id` node is used. Even if the value from this predicate is not used, the query must include it in order to inform the plugin that the graph part of the current iteration is completed.
- `mongodb:hint`: Specifies the index to be used when executing the query by calling the MongoDB method `cursor.hint()`.
- `mongodb:collation` Accepts JSON. Specifies language-specific rules for string comparison such as rules for lettercase and accent marks. It is applied to a `mongodb:find` or an `mongodb:aggregate` query.

4.10.2.7 Batch processing

While streaming is the default and preferred process for handling of the result documents, it can lead to unexpected behavior in certain edge cases, such as queries returning no or partial results despite matching data in MongoDB. This can happen when the `mongodb:find {...}` matches multiple documents, but each of the them contains only partial data. Without batch processing, when the second part of the query is evaluated, it filters out the documents that does not match the provided statements, thus resulting in unexpected behavior.

Using batch processing prevents this issue. When enabled, batch processing will collect the documents matching the input query up to the defined maximum and load them in-memory, before returning them as named graph collection.

Batch processing can be enabled via the additional MongoDB plugin predicate `mongodb:batchSize`, which controls the maximum amount of documents that will be loaded:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>

SELECT ?creativeWork ?modified WHERE {
        ?search a mongodb-index:spb1000 ;
                mongodb:find '{"@graph.cwork:audience.@id" : "cwork:NationalAudience"}' ;
                mongodb:batchSize 1000 ;
                mongodb:entity ?entity .
        GRAPH mongodb-index:spb1000 {
                ?creativeWork cwork:dateModified ?modified .
        }
}
```

In addition to the query predicate, there is also a system property called `graphdb.mongodb.maxBatchSize`, which controls the maximum value that can be set to the query predicate. The default value for the configuration is `1000`, with a maximum possible value of `10000`. These limits aim to minimize the risk of memory explosion and OOM errors in GraphDB.

4.10.2.8 Multiple index calls in the same query

The same query can support multiple MongoDB calls. There are two approaches:

- Each index call to be in a separate SUBSELECT (Example 1);
- Each index call to use different named graph. If querying different indexes, this comes out of the box. If not, use the `:graph` predicate. (Example 2).

Example 1:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
    {
        SELECT ?creativeWork ?modified {
            ?search a mongodb-index:spb1000 ;
                mongodb:find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
                mongodb:entity ?creativeWork .
            GRAPH mongodb-index:spb1000 {
                ?creativeWork cwork:dateModified ?modified ;
            }
        }
    }
    UNION
    {
        SELECT ?creativeWork ?modified WHERE {
            ?search a mongodb-index:spb1000 ;
                mongodb:find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
                mongodb:entity ?entity .
            GRAPH mongodb-index:spb1000 {
                ?creativeWork cwork:dateModified ?modified ;
            }
        }
    }
}
```

Example 2:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
    {
        ?search a mongodb-index:spb1000 ;
                mongodb:graph mongodb:search1 ;
                mongodb:find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
                mongodb:entity ?creativeWork .
        GRAPH mongodb:search1 {
                ?creativeWork cwork:dateModified ?modified ;
        }
    }
    UNION
    {
        ?search a mongodb-index:spb1000 ;
                mongodb:graph mongodb:search2 ;
```

(continues on next page)

(continued from previous page)

```
                mongodb:find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
                mongodb:entity ?entity .
        GRAPH mongodb:search2 {
                ?creativeWork cwork:dateModified ?modified ;
        }
    }
}
```

Both examples return the same result.

	creativeWork	modified
1	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/2#id	"2011-11-27T13:17:29.243+02:00"^^xsd:dateTime

4.10.2.9 Using aggregation functions

MongoDB supports a number of aggregation functions such as `min`, `max`, and `size`. These functions are called using the `:aggregate` predicate. The data of the retrieved results must be converted to an RDF model. The example below shows how to retrieve the RDF context of a MongoDB document.

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?s ?o {
    ?search a mongodb-index:spb1000 ;
        mongodb:aggregate '''[{"$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}},
                {'$addFields': {'@graph.cwork:graph.@id' :  '$@id'}}]''' ;
        mongodb:entity ?entity .
    GRAPH mongodb-index:spb1000 {
        ?s cwork:graph ?o .
    }
}
```

The `$addFields` phrase adds a new nested document in the JSON-LD stored in MongoDB. The newly added document is then parsed to the following RDF statement:

```
<http://www.bbc.co.uk/things/1#id> cwork:graph <http://www.bbc.co.uk/context/1#id>
```

We retrieve the context of the document using the `cwork:graph` predicate.

This approach is very flexible but is prone to error. Let's look at an example:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?g1 ?g2 {
    ?search a mongodb-index:spb1000 ;
        mongodb:aggregate '''[{"$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}},
                {'$addFields': {'@graph.inst:graph.@id' :  '$@id'}}]''' ;
        mongodb:entity ?entity .
    GRAPH mongodb-index:spb1000 {
        OPTIONAL {
            ?s mongodb-index:graph ?g1 .
```

(continues on next page)

(continued from previous page)

```
        }
        ?s mongodb-index:graph ?g2 .
    }
}
```

It looks very similar to the first one except that instead of `@graph.cwork:graph.@id` we are writing the value to `@graph.inst:graph.@id`, so `?g1` will not get bound. This happens because in the JSON-LD stored in MongoDB we are aware of the `cwork` context but not of the `inst` context, so `?g2` will get bound instead.

4.10.2.10 Custom fields

The following demonstrates the use of custom fields:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>

SELECT ?size ?halfSize {
    ?search a mongodb-index:spb1000 ;
        mongodb:aggregate '''[{"$match": {"@graph.@type": "cwork:NewsItem"}},
                {"$count": "size"},
                {"$project": {"custom.size": "$size", "custom.halfSize": {"$divide": ["$size", 2]}}}]''
↪' ;
        mongodb:entity ?entity .
    GRAPH mongodb-index:spb1000 {
        ?s mongodb-index:size ?size ;
        mongodb-index:halfSize ?halfSize .
    }
}
```

	size	halfSize
1	342	171.0

The values are projected as child elements of a custom node. After JSON-LD is taken from MongoDB, a preprocessing step follows in order to retrieve all child elements of custom and create statements with predicates in the `<http://www.ontotext.com/connectors/mongodb/instance#>` namespace.

Note: The returned values are always string literals.

4.10.2.11 Authentication

You can take advantage of all types of authentication by setting the credentials in the connection string. However, because it is not a good practice to store the passwords in plain text, the `:user`, `:password`, and `:authDb` predicates are available. If you use one of those predicates, you must set the other two as well. These predicates set credentials for SCRAM and LDAP authentication and the password is stored encrypted with a symmetrical algorithm on the disk. For x.509 and Kerberos authentication the connection string should be used, because no passwords are being stored.

4.11 Text mining plugin

The GraphDB text mining plugin lets you consume the output of text mining APIs as SPARQL binding variables. Depending on the annotations returned by the concrete API, the plugin makes it possible to:

- Generate semantic annotations by linking fragments from texts to knowledge graph entities (entity linking)
- Transform and filter the text annotations to a concrete RDF data model using SPARQL
- Enrich the knowledge graph with additional information suggested by the information extraction or invalidate their input
- Evaluate and control the quality of the text annotations by comparing different versions
- Implement complex text mining use cases when combined with the *Kafka GraphDB connector*

The plugin supports the protocols of these services:

- *spaCy server*
- *GATE Cloud*
- Ontotext's *Tag API*

You can also use any text mining service that outputs JSON responses as long as you provide a JSLT transformation to remodel the output from the service output to an output that the plugin understands. See the examples below for querying the *Google Cloud Natural Language API* and the *Refinitiv API* using the generic client.

4.11.1 Usage examples

A typical use case would be analyzing a piece of text such as a news story to recognize people, organizations, and locations fragments. Ideally, we will link them to entity IRIs that are already known in the knowledge graph such as Wikidata or PermID IRIs, which provide further possibilities for graph enrichment.

Let's say we have the following text that mentions Dyson as the company "Dyson Ltd.", the person "James Dyson", and also only as "Dyson".

"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state. This comes short before the founder James Dyson announced he is moving back to the UK after moving residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company."

The remaining sections of this page demonstrate how different services can annotate this text.

Note: Keep in mind that query results shown below may vary depending on the service.

4.11.1.1 spaCy server

The spaCy server is a containerized HTTP API that provides industrial-strength natural language processing whose named entity recognition (NER) component is used by the plugin.

The NER pipeline is the only spaCy component currently supported by the text mining plugin.

Creating a spaCy client

1. Run the spaCy server through its Docker image with the following commands:

```
docker pull neelkamath/spacy-server:2-en_core_web_sm-sense2vec
docker run --rm -p 8000:8000 neelkamath/spacy-server:2-en_core_web_sm-sense2vec
```

2. In the Workbench SPARQL editor, execute the following query, where `http://localhost:8000` is the location of the spaCy server that was set up using the above Docker image:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:localSpacy txtm:connect txtm:Spacy;
                    txtm:service "http://localhost:8000" .
}
```

Note: The sense2vec similarity feature is enabled by default. If your Docker image does not support it or you want to disable it when creating the client, set it to `false` in the SPARQL query:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:localSpacy txtm:connect txtm:Spacy;
                    txtm:service "http://localhost:8000";
                    txtm:sense2vec "false" .
}
```

Finding spaCy entities through GraphDB

The simplest query will return all annotations with their types and offsets. Since spaCy also provides sentence grouping, we can also get the text where each annotation was found.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?annotationText ?sentence ?annotationType ?annotationStart ?annotationEnd
WHERE {
    ?searchDocument a txtm-inst:localSpacy;
                     txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than half
↪the recruits in its headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company''' .
    graph txtm-inst:localSpacy {
        ?annotatedDocument txtm:annotations ?annotation .
        ?annotation txtm:annotationText ?annotationText ;
                txtm:annotationKey ?annotationKey;
                txtm:annotationType ?annotationType ;
                txtm:annotationStart ?annotationStart ;
                txtm:annotationEnd ?annotationEnd ;
                optional {
            ?annotation txtm:hasSentence/txtm:sentenceText ?sentence.
        }
    }
}
```

We see that spaCy succeeds in assigning the correct types to each “Dyson” found in the text:

	annotationText	annotationText	annotationType	annotationStart	annotationEnd
1	"Dyson Ltd."	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. "	"ORG"	"0"^^xsd:long	"10"^^xsd:long
2	"450"	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. "	"CARDINAL"	"25"^^xsd:long	"28"^^xsd:long
3	"more than half"	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. "	"CARDINAL"	"51"^^xsd:long	"65"^^xsd:long
4	"Singapore"	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. "	"GPE"	"102"^^xsd:long	"111"^^xsd:long
5	"250"	"The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state."	"CARDINAL"	"185"^^xsd:long	"188"^^xsd:long
6	"James Dyson"	"This comes short before the founder James Dyson announced he is moving back to the UK after moving residency to Singapore."	"PERSON"	"254"^^xsd:long	"265"^^xsd:long

Each service attaches its own metadata, which can be obtained through the feature predicate, to the annotations. In spaCy's case, we can reach the sense2vec similarity data using the following query:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?annotationText ?sentence ?annotationType ?annotationStart ?annotationEnd ?feature ?value ?
↪featureItem ?featureValue
WHERE {
    ?searchDocument a txtm-inst:localSpacy;
                    txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than half the
↪recruits in its headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company''' .
    graph txtm-inst:localSpacy {
        ?annotatedDocument txtm:annotations ?annotation .
        ?annotation txtm:annotationText ?annotationText ;
                txtm:annotationType ?annotationType ;
                txtm:annotationStart ?annotationStart ;
                txtm:annotationEnd ?annotationEnd ;
                optional {
            ?annotation txtm:hasSentence/txtm:sentenceText ?sentence.
        }
        optional {
            ?annotation txtm:features ?item .
            ?item ?feature ?value .
            optional {
                ?value ?featureItem ?featureValue .
            }
        }
    }
}
```

spaCy's sense2vec similarity feature provides us with the additional knowledge that Dyson is somehow related to "vacuums" and "Miele".

125	"Dyson"	"Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company"	"PERSON"	"341"^^xsd:long	"346"^^xsd:long	http://www.ontotext.com/textmining#sense2vec	_:164c8da7d4ac986d	http://www.ontotext.com/textmining#phrase	"Miele"
126	"Dyson"	"Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company"	"PERSON"	"341"^^xsd:long	"346"^^xsd:long	http://www.ontotext.com/textmining#sense2vec	_:164c8da7d4ac986d	http://www.ontotext.com/textmining#similarity	"0.7469000220298767"^^xsd:double
127	"Dyson"	"Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company"	"PERSON"	"341"^^xsd:long	"346"^^xsd:long	http://www.ontotext.com/textmining#sense2vec	_:164c8da7d4ac986b	http://www.ontotext.com/textmining#phrase	"vacuums"
128	"Dyson"	"Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company"	"PERSON"	"341"^^xsd:long	"346"^^xsd:long	http://www.ontotext.com/textmining#sense2vec	_:164c8da7d4ac986b	http://www.ontotext.com/textmining#similarity	"0.6833999752998352"^^xsd:double
129	"Dyson"	"Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company"	"PERSON"	"341"^^xsd:long	"346"^^xsd:long	http://www.ontotext.com/textmining#sense2vec	_:164c8da7d4ac9869	http://www.ontotext.com/textmining#phrase	"Miele"

4.11.1.2 GATE Cloud

The GATE Cloud text analytics service provides several pipelines to choose from. Its ANNIE named entity recognizer used by the plugin identifies basic entity types such as Person, Location, Organization, Money amounts, Time and Date expressions.

Creating a GATE client

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:gateService txtm:connect txtm:Gate;
                      txtm:service "https://cloud-api.gate.ac.uk/process-document/annie-named-entity-
↪recognizer?annotations=:Address&annotations=:Date&annotations=:Location&annotations=:Organization&
↪annotations=:Person&annotations=:Money&annotations=:Percent&annotations=:Sentence" .
}
```

You can provide the annotation types you are interested in using the query parameters.

Finding GATE entities through GraphDB

The following query uses the **`txtm:text`** predicate to pass the text that is to be analyzed to the GATE service.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?annotationText ?annotationType ?annotationStart ?annotationEnd ?feature ?value
WHERE {
        ?searchDocument a txtm-inst:gateService;
                        txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than
↪half the recruits in its headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company''' .
```

(continues on next page)

(continued from previous page)

```
    graph txtm-inst:gateService {
        ?annotatedDocument txtm:annotations ?annotation .

        ?annotation txtm:annotationText ?annotationText ;
            txtm:annotationType ?annotationType ;
            txtm:annotationStart ?annotationStart ;
            txtm:annotationEnd ?annotationEnd ;
        optional { ?annotation txtm:features ?item . ?item ?feature ?value }
    }
}
```

In GATE, sentences are returned as annotations, so they will appear as annotations in the response.

26	"Dyson Ltd"	"Organization"	"1"^^xsd:long	"10"^^xsd:long	txtm:orgType	"unknown"
27	"Dyson Ltd"	"Organization"	"1"^^xsd:long	"10"^^xsd:long	txtm:ruleFinal	"OrgFinal"
28	"Dyson Ltd"	"Organization"	"1"^^xsd:long	"10"^^xsd:long	txtm:rule	"OrgXEnding"
29	"Dyson Ltd"	"Organization"	"1"^^xsd:long	"10"^^xsd:long	txtm:matches	"243"^^xsd:int
30	"Dyson Ltd"	"Organization"	"1"^^xsd:long	"10"^^xsd:long	txtm:matches	"250"^^xsd:int
31	"Dyson"	"Organization"	"343"^^xsd:long	"348"^^xsd:long	txtm:NMRule	"Unknown"
32	"Dyson"	"Organization"	"343"^^xsd:long	"348"^^xsd:long	txtm:kind	"PN"
33	"Dyson"	"Organization"	"343"^^xsd:long	"348"^^xsd:long	txtm:rule	"Unknown"
34	"Dyson"	"Organization"	"343"^^xsd:long	"348"^^xsd:long	txtm:matches	"243"^^xsd:int
35	"Dyson"	"Organization"	"343"^^xsd:long	"348"^^xsd:long	txtm:matches	"250"^^xsd:int
36	"James Dyson"	"Person"	"256"^^xsd:long	"267"^^xsd:long	txtm:ruleFinal	"PersonFinal"
37	"James Dyson"	"Person"	"256"^^xsd:long	"267"^^xsd:long	txtm:kind	"fullName"
38	"James Dyson"	"Person"	"256"^^xsd:long	"267"^^xsd:long	txtm:rule	"PersonJobTitle"
39	"US$29 billion"	"Money"	"392"^^xsd:long	"405"^^xsd:long	txtm:kind	"number"
40	"US$29 billion"	"Money"	"392"^^xsd:long	"405"^^xsd:long	txtm:rule	"MoneyUnitSymbol"
41	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore.	"Sentence"	"0"^^xsd:long	"113"^^xsd:long		
42	"\nThe company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state."	"Sentence"	"113"^^xsd:long	"219"^^xsd:long		
43	"This comes short before the founder James Dyson announced he is moving back to the UK after moving residency to Singapore."	"Sentence"	"220"^^xsd:long	"342"^^xsd:long		

4.11.1.3 Tag

Ontotext's Tag API lets you semantically enrich content with annotations about both known and novel concepts.

Using machine learning algorithms and data from DBpedia and Wikidata, it recognizes mentions of entities such as Person, Organization, and Location, various relationships between them, and general topics and key phrases. Visit the NOW demonstrator to explore such entities found in news.

Creating a Tag client

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:tagService txtm:connect txtm:Ces;
                    txtm:service "https://tagger.ontotext.com/extract" ;
                    txtm:header "X-API-Key: qQ-H4f>.G}9Tqbfi&=%i" .
}
```

Note: The API key shown above is for demo purposes and will return a limited number of results in the course of a day. Contact your Ontotext representative about obtaining your own API key if you are interested in using the Tag service as part of your own applications.

Finding Tag entities through GraphDB

The following query uses the **`txtm:text`** predicate to pass the text to be analyzed to the Tag service:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?annotationText ?annotationType ?annotationStart ?annotationEnd ?feature ?value
WHERE {
    ?searchDocument a txtm-inst:tagService;
                        txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than half
↪the recruits in its headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company.''' .
    graph txtm-inst:tagService {
        ?annotatedDocument txtm:annotations ?annotation .
        ?annotation txtm:annotationText ?annotationText ;
                txtm:annotationType ?annotationType ;
                txtm:annotationStart ?annotationStart ;
                txtm:annotationEnd ?annotationEnd ;
                {
            ?annotation txtm:features ?item .
            ?item ?feature ?value
        }
    }
}
```

For some annotations, it finds an exact match to one or more IRIs in the knowledge graph and provides annotation data in the annotation variables.

92	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:isGenerated	"true"
93	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:exactMatch	http://dbpedia.org/resource/James_
94	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:relevanceScore	"1.0"^^xsd:double
95	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:inst	http://ontology.ontotext.com/resourc
96	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:confidence	"0.7306335489863898"^^xsd:double
97	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:class	http://ontology.ontotext.com/taxono
98	"Dyson"	"Person"	"341"^^xsd:long	"346"^^xsd:long	txtm:isTrusted	"true"
99	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:isGenerated	"true"
100	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:relevanceScore	"0.34963325183374083"^^xsd:double
101	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:inst	http://data.ontotext.com/publishing/
102	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:confidence	"0.5"^^xsd:double
103	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:class	http://ontology.ontotext.com/taxono
104	"250 engineers"	"Keyphrase"	"185"^^xsd:long	"198"^^xsd:long	txtm:isTrusted	"false"
105	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:isGenerated	"true"
106	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:exactMatch	http://dbpedia.org/resource/James_
107	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:relevanceScore	"1.0"^^xsd:double
108	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:inst	http://ontology.ontotext.com/resourc
109	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:confidence	"0.8484217708382679"^^xsd:double
110	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:class	http://ontology.ontotext.com/taxono
111	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	txtm:isTrusted	"true"

Tag also succeeds in assigning the proper type "Person" for "Dyson".

Here are some details about the features that Tag provides for each annotation:

- `txtm:inst`: The ID of the concept from the knowledge graph which was assigned to this annotation, or an ID of a generated concept in case it is not trusted (see `txtm:isTrusted` below).

  For example, http://ontology.ontotext.com/resource/9cafep — you can find a short description and news that mention this entity in the NOW web application at http://now.ontotext.com/#/concept&uri=http://ontology.ontotext.com/resource/9cafep, using the IRI value as uri parameter.

- `txtm:class`: The class of the concept from the knowledge graph which was assigned to this annotation.
- `txtm:isTrusted`: Has value `true` when the entity is mapped to an existing entity in the database.
- `txtm:isGenerated`: Has value `true` when the annotation has been generated by the pipeline itself — that is, from NER taggers for which there is no suitable concept in the knowledge graph. Note that generation of this does not mean that the annotation is not trusted.
- `txtm:relevanceScore`: A floating point number that represents the level of relevancy of the annotation to the target document.
- `txtm:confidence`: A floating point number that represents the confidence score for the annotation to be produced.

Extracting Tag entities using the Web Annotation Data Model

The Tag service provides a way to serve entities and their features as RDF. The model is based on the Web Annotation Data Model. The following headers should be passed when creating the Tag client:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:tagInstJSONLD txtm:connect txtm:Ces;
                 txtm:service "https://tagger.ontotext.com/extract";
                 txtm:header "Accept: application/vnd.ontotext.ces+json+ld";
                 txtm:header "Content-type: application/vnd.ontotext.ces+json+ld" ;
                 txtm:header "X-API-Key: qQ-H4f>.G}9Tqbfi&=%i" .
}
```

The common model applied for all services is no longer used because you get the Tag response in RDF as it is formed by the service.

The following request type (`Content-type`) and response type (`Accept`) combinations are supported:

- `Content-type: text/plain` — `Accept: application/vnd.ontotext.ces+json` (the default if nothing is specified)
- `Content-type: application/vnd.ontotext.ces+json+ld` — `Accept: application/vnd.ontotext.ces+json`
- `Content-type: application/vnd.ontotext.ces+json+ld` — `Accept: application/vnd.ontotext.ces+json+ld`

Not supported:

- `Content-type: text/plain` — `Accept: application/vnd.ontotext.ces+json+ld`
- `Content-type: application/vnd.ontotext.ces+json`

Note: This means that JSON-LD as a response type requires that the request is JSON-LD and nothing else. The default `text/plain` will not work, so when creating the plugin, you need to pass the `Content-type` explicitly.

When the request type is JSON-LD, the response type can be JSON or JSON-LD.

When using JSON-LD, the following document features are required. They must be passed using the `txtm:features` predicate on `?annotatedDocument` and in the following order:

```
txtm:features (?id ?title ?type ?author ?source ?category ?date).
```

Here is a sample query:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX resource: <http://ontology.ontotext.com/resource/>
PREFIX content: <http://data.ontotext.com/content/>
PREFIX onto: <http://www.ontotext.com/>
CONSTRUCT { ?subject ?predicate ?object }
WHERE {
        ?searchDocument a txtm-inst:tagInstJSONLD;
                        txtm:features (resource:guid-for-the-annotated-document "Dyson Ltd. hires
↪450 people globally" "Article" "The author"  <https://the_doc_source_uri> content:My_Category "2019-
↪03-01T00:11:15Z");
                        txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than
↪half the recruits in its headquarters in Singapore. The company best known for its vacuum cleaners
↪and hand dryers will add 250 engineers in the city-state. This comes short before the founder James
↪Dyson announced he is moving back to the UK after moving residency to Singapore. Dyson, a prominent
↪Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating
↪his company. ''' ;
    graph txtm-inst:tagInstJSONLD {
        ?subject ?predicate ?object
    }
}
```

50	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	rdf:type	http://ontology.ontotext.com/content#Article
51	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#author	"The author"
52	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#category	"http://data.ontotext.com/content/My_Category"
53	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#content	"Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore. The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state. This comes short before the founder James Dyson announced he is moving back to the UK after moving residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his company. "
54	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#publishDate	"2019-03-01T00:11:15Z"^^xsd:dateTime
55	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#source	https://the_doc_source_uri
56	http://ontology.ontotext.com/resource/guid-for-the-annotated-document	http://ontology.ontotext.com/content#title	"Dyson Ltd. hires 450 people globally"
57	http://ontology.ontotext.com/resource/8be773c2-5e24-4470-bb8d-52e39c316063	rdf:type	http://www.w3.org/ns/oa#TimeState
58	http://ontology.ontotext.com/resource/8be773c2-5e24-4470-bb8d-52e39c316063	http://www.w3.org/ns/oa#sourceDate	"2019-03-01T00:11:15Z"^^xsd:dateTime
59	http://ontology.ontotext.com/resource/c6070f2b-f165-4825-b6a4-39206b22f3e9	rdf:type	http://www.w3.org/ns/oa#Annotation
60	http://ontology.ontotext.com/resource/c6070f2b-f165-4825-b6a4-39206b22f3e9	rdf:type	http://ontology.ontotext.com/annotation#CurationAnnotation
61	http://ontology.ontotext.com/resource/c6070f2b-f165-4825-b6a4-39206b22f3e9	http://www.w3.org/ns/oa#hasBody	http://ontology.ontotext.com/resource/8707c085-28f9-494a-94cd-0b02ecf1ed4d
62	http://ontology.ontotext.com/resource/c6070f2b-f165-4825-b6a4-39206b22f3e9	http://www.w3.org/ns/oa#hasTarget	http://ontology.ontotext.com/resource/7166e80c-1dc2-4721-a034-46e395dc543e
63	http://ontology.ontotext.com/resource/c6070f2b-f165-4825-b6a4-39206b22f3e9	http://www.w3.org/ns/oa#motivatedBy	http://www.w3.org/ns/oa#tagging
64	http://ontology.ontotext.com/resource/8707c085-28f9-494a-94cd-0b02ecf1ed4d	rdf:type	http://ontology.ontotext.com/annotation#Keyphrase
65	http://ontology.ontotext.com/resource/8707c085-28f9-494a-94cd-0b02ecf1ed4d	http://ontology.ontotext.com/annotation#annotationSetName	"Final"
66	http://ontology.ontotext.com/resource/8707c085-28f9-494a-94cd-0b02ecf1ed4d	http://ontology.ontotext.com/annotation#class	http://ontology.ontotext.com/taxonomy/Thing

You can also use the `txtm:rawInput` predicate to provide your own raw JSON-LD document. Revising the query above to do this, it will look like the following, and will return the same results:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX resource: <http://ontology.ontotext.com/resource/>
PREFIX content: <http://data.ontotext.com/content/>
PREFIX onto: <http://www.ontotext.com/>
CONSTRUCT { ?subject ?predicate ?object }
WHERE {
        ?searchDocument a txtm-inst:tagInstJSONLD;
                        txtm:rawInput '''{
     "@id": "resource:some-new-guid-for-the-annotated-document-resource",
```

(continues on next page)

(continued from previous page)

```
    "@graph": [
      {
        "@id": "resource:some-new-guid-for-the-annotated-document-resource",
        "@type": "AnnotatedDocument",
        "document": {
          "@id": "http://ontology.ontotext.com/resource/guid-for-the-annotated-document",
          "@type": "Article",
          "author": "The author",
          "documentSource": "https://the_doc_source_uri",
          "category": "http://data.ontotext.com/content/My_Category",
          "publishDate": "2019-03-01T00:11:15Z",
          "title": "Dyson Ltd. hires 450 people globally",
          "docContent": "Dyson Ltd. plans to hire 450 people globally, with more than half the
↪recruits in its headquarters in Singapore. The company best known for its vacuum cleaners and hand
↪dryers will add 250 engineers in the city-state. This comes short before the founder James Dyson
↪announced he is moving back to the UK after moving residency to Singapore. Dyson, a prominent Brexit
↪supporter who is worth US$29 billion, faced criticism from British lawmakers for relocating his
↪company. "
        }
      }
    ],
    "@context": [
      "http://www.w3.org/ns/anno.jsonld",
      {
        "ann": "http://data.ontotext.com/annotation/",
        "ontoa": "http://ontology.ontotext.com/annotation#",
        "ontocontent": "http://ontology.ontotext.com/content#",
        "onto": "http://ontology.ontotext.com/taxonomy/",
        "content": "http://data.ontotext.com/content/",
        "resource": "http://ontology.ontotext.com/resource/",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "nif": "http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#",
        "Article": "ontocontent:Article",
        "AnnotatedDocument": "ontocontent:AnnotatedDocument",
        "document": "ontocontent:document",
        "annotations": "ontocontent:annotations",
        "author": {
          "@id": "ontocontent:author",
          "@type": "xsd:string"
        },
        "documentSource": {
          "@id": "ontocontent:source",
          "@type": "@id"
        },
        "category": {
          "@id": "ontocontent:category",
          "@type": "@id"
        },
        "publishDate": {
          "@id": "ontocontent:publishDate",
          "@type": "xsd:dateTime"
        },
        "title": {
          "@id": "ontocontent:title",
          "@type": "xsd:string"
        },
        "docContent": "ontocontent:content",
        "tagType": {
          "@id": "ontoa:tagType",
          "@type": "@id"
        },
```

(continues on next page)

(continued from previous page)

```
        "relevanceScore": {
          "@id": "ontoa:relevanceScore",
          "@type": "xsd:double"
        },
        "confidence": {
          "@id": "nif:confidence",
          "@type": "xsd:double"
        },
        "type": {
          "@id": "ontoa:type",
          "@type": "xsd:string"
        },
        "class": {
          "@id": "ontoa:class",
          "@type": "@id"
        },
        "status": {
          "@id": "ontoa:status",
          "@type": "xsd:string"
        },
        "isTrusted": {
          "@id": "ontoa:isTrusted",
          "@type": "xsd:boolean"
        },
        "isGenerated": {
          "@id": "ontoa:isGenerated",
          "@type": "xsd:boolean"
        },
        "annotationSetName": {
          "@id": "ontoa:annotationSetName",
          "@type": "xsd:string"
        },
        "annotationType": {
          "@id": "ontoa:type",
          "@type": "xsd:string"
        }
      }
    ]
  } ''' ;
   graph txtm-inst:tagInstJSONLD {
       ?subject ?predicate ?object
   }
}
```

The supported returned response formats are JSON and JSON-LD.

4.11.1.4 Extracting annotations from another NER service

To register a service in the text mining plugin, the service must provide a REST interface with a POST endpoint. The response `Content-Type` value must be `application/json`. Pass the POST request headers using the predicate `http://www.ontotext.com/textmining#header`. Pass the request body with the predicate `http://www.ontotext.com/textmining#text`.

The following curl request

```
curl -X POST --header "HEADER1: VALUE1" --header "HEADER2: VALUE2" -d 'body' 'https://endoint.com?
↪queryParam1=param1'
```

corresponds to the following configuration,

```
PREFIX : <http://www.ontotext.com/textmining#>
PREFIX inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    inst:myService :connect :Provider;
                       :service "https://endoint.com?queryParam1=param1";
                       :header "HEADER1: VALUE1";
                       :header "HEADER2: VALUE2";
                       :transformation '''
                       ...
                       '''.
}
```

and to the following query for consuming the annotations:

```
PREFIX : <http://www.ontotext.com/textmining#>
PREFIX inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?annotationText ?annotationType ?annotationStart ?annotationEnd ?feature ?value
WHERE {
    ?searchDocument a inst:myService;
                        :text '''body''' .
    graph inst:myService {
        ?annotatedDocument :annotations ?annotation .
        ?annotation :annotationText ?annotationText ;
                :annotationType ?annotationType ;
                :annotationStart ?annotationStart ;
                :annotationEnd ?annotationEnd ;
                {
            ?annotation :features ?item .
            ?item ?feature ?value
        }
    }
}
```

To extract annotations using another named entity recognition provider, create a client for these services using a JSLT transformation. The transformation will convert the JSON returned by the target service to a JSON model that the text mining plugin understands. The target JSON should follow this model:

```
{
   "content":"",
   "sentences":[ ],
   "features":{ },
   "annotations":[
      {
         "text":"Google",
         "type":"Company",
         "startOffset":78,
         "endOffset":84,
         "confidence":0.0,
         "features":{  }
      }
   ]
}
```

where the only required part is the following:

```
{

   "annotations":[
      {
         "text":"Google",
         "type":"Company",
```

(continues on next page)

(continued from previous page)

```
        "startOffset":78,
        "endOffset":84,
      }
    ]
}
```

Google Cloud Natural Language API

The Google Cloud Natural Language API associates information such as salience and mentions with annotations, where an annotation represents a phrase in the text that is a known entity such as a person, an organization, or a location. This also requires a token to access the API.

Creating a Google Cloud Natural Language API client

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
    txtm-inst:myGoogleService txtm:connect txtm:Provider;
                      txtm:service "https://language.googleapis.com/v1/documents:annotateText";
                      txtm:header "Authorization: Bearer <your API token>";
                      txtm:transformation '''
                      {"annotations" : flatten([for (.entities)
                                let type = .type
                                let metadata = .metadata
                                let salience = .salience
                                let mentions = [for (.mentions) {
                                    "type" : $type,
                                    "text" : .text.content,
                                    "startOffset" : .text.beginOffset,
                                    "endOffset" : .text.beginOffset + size(.text.content),
                                    "features" : {
                                          "salience" : $salience,
                                          "metadata" : $metadata
                                    }
                                  }]
                                  $mentions
                      ])}
                      '''.
}
```

Extracting entities from Google Cloud Natural Language API

Once created, you can list annotations using a model similar to the other services. Note that you need to provide the input in the way the service expects it. No transformation is applied to the request content.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?annotationText ?annotationType ?annotationStart ?annotationEnd ?feature ?value ?featureItem ?
↪featureValue
WHERE {
    ?searchDocument a txtm-inst:myGoogleService;
                      txtm:text '''
        {
        "document":{
        "type":"PLAIN_TEXT",
```

(continues on next page)

(continued from previous page)

```
        "content":"Net income was $9.4 million compared to the prior year of $2.7 million. Google is a
↪big company.
        Revenue exceeded twelve billion dollars, with a loss of $1b"
        }, "features": {'extractEntities': 'true', 'extractSyntax': 'true'},
        'encodingType':'UTF8',
        }
        ''' .
            graph txtm-inst:myGoogleService {
                ?annotatedDocument txtm:annotations ?annotation .
                ?annotation txtm:annotationText ?annotationText ;
                        txtm:annotationType ?annotationType ;
                        txtm:annotationStart ?annotationStart ;
                        txtm:annotationEnd ?annotationEnd ;
                        optional {
                    ?annotation txtm:features ?item .
                    ?item ?feature ?value .
                    optional { ?value ?featureItem ?featureValue . }
                }
            }
        }
```

The results will look like this:

	annotationText	annotationType	annotationStart	annotationEnd	feature	value	featureItem	featureValue
1	"Net income"	"OTHER"	"0"^^xsd:long	"10"^^xsd:long	:salience	"0.57806146"^^xsd:double		
2	"Google"	"ORGANIZATION"	"72"^^xsd:long	"78"^^xsd:long	:salience	"0.24941672"^^xsd:double		
3	"Google"	"ORGANIZATION"	"72"^^xsd:long	"78"^^xsd:long	:metadata	_:node1f5ig7tvvx4	:mid	"/m/045c7b"
4	"Google"	"ORGANIZATION"	"72"^^xsd:long	"78"^^xsd:long	:metadata	_:node1f5ig7tvvx4	:wikipedia_url	https://en.wikipedia.org/wiki/G
5	"company"	"ORGANIZATION"	"88"^^xsd:long	"95"^^xsd:long	:salience	"0.24941672"^^xsd:double		
6	"company"	"ORGANIZATION"	"88"^^xsd:long	"95"^^xsd:long	:metadata	_:node1f5ig7tvvx7	:mid	"/m/045c7b"
7	"company"	"ORGANIZATION"	"88"^^xsd:long	"95"^^xsd:long	:metadata	_:node1f5ig7tvvx7	:wikipedia_url	https://en.wikipedia.org/wiki/G
8	"Revenue"	"OTHER"	"105"^^xsd:long	"112"^^xsd:long	:salience	"0.07516972"^^xsd:double		
9	"loss"	"OTHER"	"153"^^xsd:long	"157"^^xsd:long	:salience	"0.057097316"^^xsd:double		
10	"b"	"OTHER"	"163"^^xsd:long	"164"^^xsd:long	:salience	"0.040254783"^^xsd:double		
11	"$9.4 million"	"PRICE"	"15"^^xsd:long	"27"^^xsd:long	:salience	"0"^^xsd:int		
12	"$9.4 million"	"PRICE"	"15"^^xsd:long	"27"^^xsd:long	:metadata	_:node1f5ig7tvvx13	:value	"9400000.000000"
13	"$9.4 million"	"PRICE"	"15"^^xsd:long	"27"^^xsd:long	:metadata	_:node1f5ig7tvvx13	:currency	"USD"
14	"$2.7 million"	"PRICE"	"58"^^xsd:long	"70"^^xsd:long	:salience	"0"^^xsd:int		
15	"$2.7 million"	"PRICE"	"58"^^xsd:long	"70"^^xsd:long	:metadata	_:node1f5ig7tvvx16	:value	"2700000.000000"
16	"$2.7 million"	"PRICE"	"58"^^xsd:long	"70"^^xsd:long	:metadata	_:node1f5ig7tvvx16	:currency	"USD"
17	"twelve billion dollars"	"PRICE"	"122"^^xsd:long	"144"^^xsd:long	:salience	"0"^^xsd:int		
18	"twelve billion dollars"	"PRICE"	"122"^^xsd:long	"144"^^xsd:long	:metadata	_:node1f5ig7tvvx19	:currency	"USD"
19	"twelve billion dollars"	"PRICE"	"122"^^xsd:long	"144"^^xsd:long	:metadata	_:node1f5ig7tvvx19	:value	"12000000000.000000"
20	"$1"	"PRICE"	"161"^^xsd:long	"163"^^xsd:long	:salience	"0"^^xsd:int		
21	"$1"	"PRICE"	"161"^^xsd:long	"163"^^xsd:long	:metadata	_:node1f5ig7tvvx22	:currency	"USD"
22	"$1"	"PRICE"	"161"^^xsd:long	"163"^^xsd:long	:metadata	_:node1f5ig7tvvx22	:value	"1.000000"
23	"9.4 million"	"NUMBER"	"16"^^xsd:long	"27"^^xsd:long	:salience	"0"^^xsd:int		
24	"9.4 million"	"NUMBER"	"16"^^xsd:long	"27"^^xsd:long	:metadata	_:node1f5ig7tvvx25	:value	"9400000.000000"

Refinitiv API

Refinitiv's PermIDs are open, permanent, and universal identifiers where underlying attributes capture the context of the identity they each represent.

Creating a Refinitiv API client

The following provides a model for creating a Refinitiv API client.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
        txtm-inst:refinitiv txtm:connect txtm:Provider;
                            txtm:service "https://api-eit.refinitiv.com/permid/calais";
                            txtm:header "X-AG-Access-Token: <your_access_token>";
                            txtm:header "Content-Type: text/raw";
                            txtm:header "x-calais-selectiveTags: company,person,industry,socialtags,topic
↪";
                            txtm:header "outputformat: application/json";
                            txtm:transformation '''
                            {
                                "content" : string(.doc.info.document),
                                "rawSource" : string(.),
                                "language" : .doc.meta.language,
                                "features" : {for (.) .key : {for (.value) .key : .value }
                                     if (.value._typeGroup and .value._typeGroup != "entities" and .value.
↪_typeGroup != "relations"
                                     and .value._typeGroup != "language" and .value._typeGroup !=
↪"versions") },
                                "annotations" : flatten([for (.)
                                 if (.value._typeGroup == "entities")
                                   let type = .value._type
                                   let text = .value.name
                                   let features = {for (.value) .key : .value
                                   if (.key != "_type" and .key != "name" and .key != "instances" and .
↪key != "offset")}

                                   let instances = [for (.value.instances){
                                       "type" : $type,
                                       "text" : $text,
                                       "startOffset": .offset,
                                       "endOffset" : .offset + size($text),
                                       "features" : $features
                                     }]
                                     $instances
                                   else if (.value._typeGroup == "relations")
                                    let type = .value._type
                                    let features = {for (.value) .key : .value
                                    if (.key != "_type" and .key != "instances")}
                                      let instances = [for (.value.instances){
                                         "type" : $type,
                                         "text" : .exact,
                                         "startOffset": .offset,
                                         "endOffset" : .offset + size(.exact),
                                         "features" : $features
                                      }]
                                      $instances
                                   else
                                       []
                                ])
```

(continues on next page)

(continued from previous page)

```
                        }
                      ''' .
    }
```

Extracting Refinitiv PermID entities

The following query uses the **`txtm:text`** predicate to pass the text to be analyzed to the Refinitiv service:

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?searchDocument ?annotation ?annotationText ?annotationType ?annotationStart ?annotationEnd ?
↪feature ?value ?featureItem ?featureValue
WHERE {
       ?searchDocument a txtm-inst:refinitiv;
                       txtm:text '''Dyson Ltd. plans to hire 450 people globally, with more than half
↪the recruits in its headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company.''' .
  graph txtm-inst:refinitiv {
     ?annotatedDocument txtm:annotations ?annotation .
     ?annotation txtm:annotationText ?annotationText ;
        txtm:annotationType ?annotationType ;
        txtm:annotationStart ?annotationStart ;
        txtm:annotationEnd ?annotationEnd ;
     optional {
        ?annotation txtm:features ?item . ?item ?feature ?value .
        optional { ?value ?featureItem ?featureValue . }
     }
  }
}
```

	searchDocument	annotation	annotationText	annotationType	annotationStart	annotationEnd	feature	value	featureItem	featureValue
1	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:_typeGroup	"entities"		
2	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:forenduserdisplay	"true"		
3	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:nationality	"N/A"		
4	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:confidencelevel	"0.997"		
5	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:persontype	"N/A"		
6	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:firstname	"James"		
7	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:lastname	"Dyson"		
8	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:commonname	"James Dyson"		
9	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:confidence	_:node1f61vlia5x28	:statisticalfeature	"0.983"
10	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:confidence	_:node1f61vlia5x28	:dblookup	"0.95"
11	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:confidence	_:node1f61vlia5x28	:resolution	"0.0"
12	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:confidence	_:node1f61vlia5x28	:aggregate	"0.997"
13	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:_typeReference	http://s.opencalais.com/		
14	_:node1f5sgetoux285804	http://www.ontotext.cor	"James Dyson"	"Person"	"254"^^xsd:long	"265"^^xsd:long	:permid	https://permid.org/1-404011		

The tricky part of the integration of an arbitrary NER provider is to write the JSLT transformation, but once you get used to the language, you can enrich your text document with any entity provider of your choice, and extend your knowledge graph solely with the power of SPARQL and GraphDB.

Escaping special characters

The following example demonstrates how quotation marks within the quoted input string can be escaped with double-backslash sequences:

```
PREFIX : <http://www.ontotext.com/textmining#>
PREFIX inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?annotationText ?annotationType ?annotationStart ?annotationEnd ?feature ?value
WHERE {
    ?searchDocument a inst:razor;
                        :text '''
{"text":"Prosecutors want NFL's Peterson arrested on alleged bond violation | Reuters
    Prosecutors want NFL's Peterson arrested on alleged bond violation
    By Eric Kelsey
    (Reuters) --- Suspended Minnesota Vikings star Adrian Peterson faced new legal trouble on Thursday
↪after Texas prosecutors in his child abuse case asked a court to order his arrest on a possible drug-
↪related bond violation.
    Peterson, 29, who has been accused of injuring his 4-year-old son while disciplining him with the
↪thin end of a tree branch, allegedly told a drug-testing administrator on Wednesday he had smoked
↪marijuana before submitting to a urinalysis test, court papers said.
    \\"During this process the defendant admitted ... that he smoked a little weed,\\" according to the
↪motion filed by Montgomery County District Attorney Brett Ligon.
    A court date has not been set on the possible bond violation. Peterson's next scheduled court date
↪is Nov. 4.
    It is unclear when a judge would rule on the motion as prosecutors' request to have the current
↪judge recused must be heard first.
    Peterson's attorney, Rusty Hardin, declined to comment until a judge is settled on in the case.
    The Vikings said in a statement they were aware of the allegation and \\"will await the results of
↪that hearing before having further comment.\\"
    The National Football League did not respond to a request for comment.
    Peterson was arrested and posted $15,000 bond on Sept. 12 on a charge of injury to a child. He was
↪later suspended indefinitely with pay by the Vikings until the matter is resolved.
    He has admitted using a switch, the thin end of a tree branch, to discipline his son, but said he
↪was not trying to injure him.
    Peterson could be sentenced to up to two years in prison and fined $10,000 if convicted.
    The charge against Peterson came as the NFL faced public criticism for its handling of a spate of
↪domestic violence cases among its players. A number of corporate sponsors rebuked America's most
↪popular professional sports league, which has overhauled how it deals with player behavior and
↪punishment.
    (Reporting by Eric Kelsey in Los Angeles; Editing by Peter Cooney )
"}
''' .
    graph inst:razor {
        ?annotatedDocument :annotations ?annotation .
        ?annotation :annotationText ?annotationText ;
                :annotationType ?annotationType ;
                :annotationStart ?annotationStart ;
                :annotationEnd ?annotationEnd ;
                {
            ?annotation :features ?item .
            ?item ?feature ?value
        }
    }
}
```

Since the text enclosed within the `'''` marks represents a literal string, SPARQL will store it as is and keep new lines and paragraphs. The only special characters that need to be escaped with a double backslash are the quotation marks: `\\"`.

4.11.1.5 Comparing annotations between services

The text mining plugin generates meaningful IRIs for the `?annotatedDocument` and `?annotation` variables. It provides the additional `txtm:annotationKey` predicate that binds an IRI for the annotation, based on the text and offsets, to the `?annotationKey` variable. This means that regardless of the service that generated the annotation, the same pieces of text will have the same `?annotationKey` IRIs. You can use this to compare annotations over the same piece of text provided by different services.

The following query compares annotation types obtained from spaCy and Tag for annotations that have the same key and text — that is, they refer to the same piece of text.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT ?spacyDocument ?tagDocument ?spacyAnnotation ?tagAnnotation ?spacyType ?tagType ?annotationKey ?
↪annotationText
WHERE {
    BIND ('''Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its
↪headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company''' as ?text)
    ?searchDocument1 a txtm-inst:localSpacy;
                     txtm:text ?text.
    graph txtm-inst:localSpacy {
        ?spacyDocument txtm:annotations ?spacyAnnotation .

        ?spacyAnnotation txtm:annotationText ?annotationText ;
                         txtm:annotationKey ?annotationKey;
                 txtm:annotationType ?spacyType .
    }

    ?searchDocument2 a txtm-inst:tagService;
                     txtm:text ?text .
    graph txtm-inst:tagService {
        ?tagDocument txtm:annotations ?tagAnnotation .
        ?tagAnnotation txtm:annotationText ?annotationText ;
                         txtm:annotationKey ?annotationKey;
                 txtm:annotationType ?tagType .
    }
}
```

The query returns the following:

	spacyDocument	tagDocument	spacyAnnotation	tagAnnotation	spacyType	tagType	annotationKey	annotationText
1	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111/localSpacy/4	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111/tagService/3	"GPE"	"Location"	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111	"Singapore"
2	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/254/265/localSpacy/6	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/254/265/tagService/13	"PERSON"	"Person"	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/254/265	"James Dyson"
3	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/330/339/localSpacy/8	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/330/339/tagService/4	"GPE"	"Location"	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/330/339	"Singapore"
4	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/341/346/localSpacy/9	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/341/346/tagService/11	"PERSON"	"Person"	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/341/346	"Dyson"
5	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/360/366/localSpacy/10	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/360/366/tagService/5	"ORG"	"Thing"	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/360/366	"Brexit"

The IRIs generated by the text mining plugin have the following meaning:

- `?annotatedDocument` (`?tagDocument` or `?spacyDocument` in the above query): `<http://www.ontotext.com/textmining/document/<md5-content>>` where `md5-content` is the MD5 code of the document content.

For example: `<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847>`.

Note that document IRIs will be the same for the same pieces of text, regardless of the service.

- `?annotation`: `<http://www.ontotext.com/textmining/document/<md5-content>/annotation/<start>/<end>/<service-name>/<index>>`
  - `<start>/<end>`: The start/end offsets of the annotation in the text.
  - `<service-name>`: The name of the service that provided the annotation.
  - `<index>`: A unique number of the annotation within the document, meaning that if there are different annotation for the same pieces of text, they will have different IRIs.

  For example:

  `<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111/localSpacy/4>`

  `<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111>`

- `?annotationKey`: `<http://www.ontotext.com/textmining/document/<md5-content>/annotation/<start>/<end>>`: The annotation key IRI marks only a piece of text in the document and can be used to find annotation over the same piece of text, but provided by different services.

  For example: `<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111>`

4.11.1.6 Enriching documents with mentions of known entities

Using the Tag `txtm:exactMatch` feature and our own mentions predicate, we can generate the following triples and enrich our dataset with entities from DBpedia.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX my-kg: <http://my.knowledge.graph.com/textmining#>
CONSTRUCT {
    ?tagDocument my-kg:mentions ?value
}
WHERE {
    BIND ('''Dyson Ltd. plans to hire 450 people globally, with more than half the recruits in its
↪headquarters in Singapore.
The company best known for its vacuum cleaners and hand dryers will add 250 engineers in the city-state.
↪ This comes short before the founder James Dyson announced he is moving back to the UK after moving
↪residency to Singapore. Dyson, a prominent Brexit supporter who is worth US$29 billion, faced
↪criticism from British lawmakers for relocating his company''' as ?text)
    ?searchDocument a txtm-inst:tagService;
                    txtm:text ?text .
    graph txtm-inst:tagService {
        ?tagDocument txtm:annotations ?tagAnnotation .
        ?tagAnnotation txtm:features ?item .
        ?item txtm:exactMatch ?value
    }
}
```

The query returns the following:

	subject	predicate	object
1	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Singapur
2	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Сингапур
3	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Singapore
4	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Plan
5	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Vereinigtes_Königreich
6	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Обединено_кралство_Великобритания_и_Северна_Ирландия
7	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/United_Kingdom
8	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Singapur
9	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Сингапур
10	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Singapore
11	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/Brexit
12	http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06	http://my.knowledge.graph.com/textmining#mentions	http://dbpedia.org/resource/EU-Austritt_des_Vereinigten_Königreichs

Of course, the power of RDF allows you to construct any graph you want based on the response from the named entity recognition service.

4.11.2 Error handling

To see some error handling in action, let's first create three documents to annotate in one batch. Use the following update request to insert the three sample documents in your repository:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX my-kg: <http://my.knowledge.graph.com/textmining#>

INSERT DATA
      {
      GRAPH <http://my.knowledge.graph.com> {
        my-kg:doc1 my-kg:content "SOFIA, March 14 (Reuters) --- Bulgaria expects Azeri state energy
↪company SOCAR to start investing in the Balkan country's retail gas distribution network this year,
↪Prime Minister said on Thursday".
        my-kg:doc2 my-kg:content "Bulgaria is looking to secure gas supplies for its planned gas hub at
↪the Black Sea port of Varna and Borissov said he had discussed the possibility of additional Azeri
↪gas shipments for the plan.".
        my-kg:doc3 my-kg:content "In the Sunny Beach resort, this one-bedroom apartment is 150m from
↪the sea. It is in the Yassen complex, which has a communal pool and gardens. On the third floor, the
↪66sq m (718sq ft) apartment has a livingroom, with kitchen, that opens to a balcony overlooking the
↪pool. There are also a bedroom and bathroom. The property is being sold with furniture. The service
↪charge is €8 a square metre, making it about €528. Burgas Airport is about 12km away. Varna is 40km
↪away.".
      }
}
```

You can send all of them for annotation with a single query. By default, if the service fails for one document, the whole query will fail. As a result, you will miss the results for the documents that were successfully annotated. To prevent this from happening, you can use the `txtm:serviceErrors` predicate that defines a maximum number of errors allowed before the query fails, where `-1` means that an infinite number of errors is allowed. As a result of the following query, you will either get an error for the document, or its annotations.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX my-kg: <http://my.knowledge.graph.com/textmining#>

SELECT ?content ?annotationText ?errorFeature
WHERE {
    ?myDocument my-kg:content ?content.
    OPTIONAL {
        ?searchDocument a txtm-inst:localSpacy;
```

(continues on next page)

(continued from previous page)

```
                        txtm:text ?content;
                        txtm:serviceErrors -1 .
        GRAPH txtm-inst:localSpacy {
            OPTIONAL {
                ?annotatedDocument txtm:annotations ?annotation .
                ?annotation txtm:annotationText ?annotationText .
            }
            OPTIONAL {
                ?annotatedDocument txtm:features ?docFeature .
                ?docFeature ?errorFeature ?errorFeature.
            }
        }
    }
}
```

The following results will be returned if the spaCy service successfully annotates the first document but is then stopped. We can simulate this by stopping the spaCy Docker during the query execution using `Ctrl+C` in the terminal where Docker is running. The error message is returned as a document feature.

	content	annotationText	errorFeature
1	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"SOFIA"	
2	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"March 14"	
3	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"Reuters"	
4	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"Bulgaria"	
5	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"Azeri state energy company"	
6	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"Balkan"	
7	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"this year"	
8	"SOFIA, March 14 (Reuters) - Bulgaria expects Azeri state energy company SOCAR to start investing in the Balkan country's retail gas distribution network this year, Prime Minister said on Thursday"	"Thursday"	
9	"Bulgaria is looking to secure gas supplies for its planned gas hub at the Black Sea port of Varna and Borissov said he had discussed the possibility of additional Azeri gas shipments for the plan."		"localhost:8000 failed to respond"
10	"Bulgaria is looking to secure gas supplies for its planned gas hub at the Black Sea port of Varna and Borissov said he had discussed the possibility of additional Azeri gas shipments for the plan."		"500"
11	"In the Sunny Beach resort, this one-bedroom apartment is 150m from the sea. It is in the Yassen complex, which has a communal pool and gardens. On the third floor, the 66sq m (718sq ft) apartment has a livingroom, with kitchen, that opens to		"localhost:8000 failed to respond"

4.11.3 Managing text mining instances

Use the queries below to explore the instances of text mining clients you have in the repository with their configurations, as well as to remove them.

4.11.3.1 Listing all clients

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT * where {
    ?instance a txtm:Service .
}
```

4.11.3.2 Getting a client's configuration

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT * WHERE {
   txtm-inst:localSpacy ?p ?o .
}
```

4.11.3.3 Dropping an instance

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
   txtm-inst:localSpacy txtm:dropService "".
}
```

4.11.4 Monitoring annotation progress

If you are annotating multiple documents in one transaction, you may want to get feedback on the progress. This is done by setting the *log level* of the text mining plugin to `DEBUG` in the `conf/logback.xml` file of the GraphDB distribution:

```
<logger name="com.ontotext.graphdb.plugins.textmining" level="DEBUG"/>
```

You will see a message for each document sent for annotation in the GraphDB main log file in the `logs` directory.

```
[DEBUG] 2021-05-19 08:39:40,893 [repositories/ff-news | c.o.g.p.t.c.ClientBase] Annotating document content starting with: "Australia's Cardinal Pell sentenced to six years jail for sexually... MELBOURNE (Reuters) - Former ..." with length: 911
```

```
[DEBUG] 2021-05-19 08:39:41,851 [repositories/ff-news | c.o.g.p.t.c.ClientBase] Annotating document content starting with: "Google engineer calls for accessibility in design Laura D'Aquila, a software engineer at Google and..." with length: 2455
```

```
[DEBUG] 2021-05-19 08:39:45,610 [repositories/ff-news | c.o.g.p.t.c.ClientBase] Annotating document content starting with: "NBA Fines Russell Westbrook $25,000, Utah Jazz Permanently Bans Fan Following Verbal Altercation Op..." with length: 4932
```

4.12 Embedding model configuration for vector search

Both *Elasticsearch GraphDB Connector* and *OpenSearch GraphDB Connector* support integrated vector search, which uses an embedding model to generate vectors and provide better recall than full-text search. You can use a variety of models to generate the embeddings, including OpenAI embedding models, Elasticsearch (when using Elasticsearch), or locally deployed models. Before you can create a connector instance with vector search capabilities, you need to first choose and configure the embedding model.

Note: This does not apply if you are using Elasticsearch to create the embeddings. Refer to the *Elasticsearch documentation for more information on how to configure the creation of the embeddings* in that scenario.

The Embedding Model Clients can also read a set of system properties configured through GraphDB. These properties can configured through GraphDB, but are described in the configuration section of the Embedding Model Clients project.

4.12.1 Using a local embedding model deployed with Graphwise Transformer

You can use a locally deployed model to generate the embeddings, allowing for advanced customization and increased security. In fact, GraphDB is distributed with a client for such a locally deployed Hugging Face transformer model, which can be deployed using the Graphwise Transformer tool.

The github page of the Graphwise Transformer provides thorough instructions on how to deploy the model using Graphwise Transformer.

Once you have deployed the model, when creating the connector instance you need to configure the `embeddingModel` parameter to `com.ontotext.embeddings.GraphwiseTransformerClient`. By default both the transformer and the client are configured with the `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` model. To change the default model used by Graphwise Transformer:

1. Set the `default_model` property in the `config.properties` file of the Graphwise Transformer to your `<model_name>`, as described in the configuration section of the Graphwise Transformer documentation.
2. Set the GraphDB configuration property `graphwise.transformer.embedding.model.name` to `<model_name>` before starting GraphDB.

Note: The default model `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` has a limit of 128 tokens. Text above that limit will be automatically truncated by the model. The Graphwise Transformer client will also truncate texts for embeddings larger than 4 megabytes because bigger documents will not fit GRPC limitations. If you have texts for embeddings bigger than these limitations, make sure you apply document chunking and create vector fields over the chunks of the documents.

Tip: If you are using a locally deployed embedding model, consider CUDA cards for faster generation. The model will automatically be placed on the most performant available device.

4.12.2 Using an OpenAI embedding model

You can also use an OpenAI embedding model to create the embeddings for vector search.

When using OpenAI to create the embeddings, the following GraphDB configuration properties must be configured prior to starting GraphDB:

- `openai.embedding.model.name` — the name of the OpenAI model you want to use for the embeddings
- `openai.embedding.model.api.key` — your OpenAI API key
- `openai.embedding.model.dimensions` — the embedding model dimensions

Then, when creating the GraphDB *Elasticsearch GraphDB Connector* and *OpenSearch GraphDB Connector*, configure the `embeddingModel` property (or the *Embedding model* if using the GraphDB Workbench) to `com.ontotext.embeddings.OpenAiEmbeddingModel`.

4.12.3 Using an alternative model provider through the EmbeddingModel interface of langchain4j

You can use an alternative model provider by implementing the EmbeddingModel interface of langchain4j. To do so, package the implementation and all its dependencies and put it in the connector classpath (`.../lib/plugins/elasticsearch-connector/`).

When you create the connector in GraphDB, specify the fully classified class name of the implementation class in Java using the embedding model field of the connector.

Tip: You can base your solution on the examples in our Embedding Model clients project.

CHAPTER

FIVE

CLUSTERS

5.1 Overview of clusters

Note: This feature requires a GraphDB Enterprise license.

A key to GraphDB's scalability is its ability to run clusters. A cluster is a collection of interconnected GraphDB instances, each assigned to a unique URL address and running on a separate machine, that work together as a coherent group. With this arrangement, a GraphDB cluster can survive the failures of some of its members.

You can create a cluster *through the GraphDB Workbench* or by *using the REST API*. The Workbench also provides a *Cluster view*, which visualizes the current state of the cluster and its nodes.

The cluster is an essential tool for building a reliable large-scale system and achieving high availability. A cluster deployment has high uptime, recovers smoothly from failures, greatly reduces the chance of data loss, and overall, handles and adapts to unexpected situations and scenarios.

In order for a cluster to process INSERT and DELETE operations, more than half of its nodes — in other words, a majority quorum of nodes — should be alive (that is, up and running). If 50% or more of the nodes are dead, the cluster will not be able to process INSERT or DELETE operations. You can learn more about this in the *Quorum-based replication* section of the cluster documentation.

See also:

- *Using a cluster* provides an overview of using the GraphDB client API to manage your cluster.
- *Cluster states and election procedures* expands on the concepts introduced in this section.
- *Creating and managing a cluster* details how to deploy a cluster in GraphDB using the Workbench or via the REST API.
- *Multi-region cluster deployments* details how to configure multiple clusters in different zones to achieve high availability in multi-region deployments.

5.1.1 The Raft consensus algorithm

The GraphDB cluster is based on a coordination mechanism known as the Raft consensus algorithm. Raft allows a collection of machines to work as a coherent group and survive *failure of some of its members*.

The algorithm implements consensus by first electing a leader and then assigning it full responsibility for managing the replicated log that makes the node redundancy possible. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply the actions described by each log entry to their state machines, which are used to keep track of the node states. It can tolerate $n \geq 2m + 1$ failures. Because of this, Raft plays a key role in achieving an enterprise-level high availability GraphDB cluster deployment.

GraphDB modifies and extends the original Raft protocol to include several improvements that are necessary for a higher level of cluster consistency. However, the leader election and automatic failover mechanism provided by Raft are implemented without changes:

- An update is considered committed when a majority of GraphDB instances actually commit it locally rather than simply storing it in their transaction logs.
- Followers also verify with the Leader whether a received entry is processed by the rest of cluster.
- An additional `OUT_OF_SYNC` state has been introduced for faulty nodes that require recovery from other nodes with correct state inside the cluster.
- Node snapshot replication, leveraging Raft's log index and replication mechanism, has been introduced to recover out-of-sync nodes.

5.1.1.1 Cluster roles

The total amount of leader and follower nodes is usually an odd number in order to tolerate failures. At any given time, each of the nodes is in one of four states:

- **Leader**: Usually, there is one leader that handles all client requests. For example, if a client contacts a follower, the follower redirects it to the leader.
- **Follower**: A cluster is made up of one leader and all other servers are followers. They are passive, meaning that they issue no requests on their own but simply respond to requests from leaders and candidates.
- **Candidate**: This state is used when electing a new leader.
- **Restricted**: In this state, the node cannot respond to requests from other nodes and cannot participate in election. A node goes into this state when there is a license issue — for example, an invalid or expired license.

Note: To learn more about the different states a node can take, see the *Cluster states* section of the documentation.

5.1.2 Internal and external proxy

5.1.2.1 Internal proxy

In normal working conditions, the cluster nodes have two states: **leader** and **follower**. The follower nodes can accept read requests, but cannot write any data. To make it easier for the user to communicate with the cluster, an integrated proxy will redirect all requests (with some exceptions) to the leader node. This ensures that regardless of which cluster node is reached, it can accept all user requests.

However, if a GraphDB cluster node is unavailable, another node on another URL must take its place. This means that all cluster node addresses must be tracked and the reached node must be healthy and online.

5.1.2.2 External proxy

For better usability, you can deploy the proxy separately on its own URL. This way, you do not need to know where all cluster nodes are. Instead, there is a single URL that will always point to the leader node.

The externally deployed proxy will behave like a regular GraphDB instance, including opening and using the Workbench. It will always know which one is currently the leader and will always serve all requests to that leader.

5.1.3 High availability and fault-tolerance

A system can be characterized as having high availability if it meets criteria such as having high uptime, smooth recovery, zero data loss, and the ability to handle and adapt to unexpected situations and scenarios. The Raft consensus algorithm helps to ensure these criteria and achieve enterprise-grade high availability deployment.

5.1.3.1 What is consensus?

One of the main problems of distributed systems is achieving consensus between its servers — in other words, getting the multiple individual machines that are part of the deployment to agree on values. Once those machines agree on a value, that decision is final. With most typical consensus algorithms, a deployment can continue operating as long as a majority of its servers are available. For example, a cluster of five servers will continue working as long as at least three of its servers are operational. However, if three or more of the servers fail, the cluster will stop making INSERT and DELETE operations, although it will never return an incorrect value.

Consensus algorithms aim to be fault-tolerant, where faults can be classified in two categories:

- **Crash failure:** The component abruptly stops functioning and does not resume. The other components can detect the crash and adjust their local decisions in time.
- **Byzantine failure:** The component behaves arbitrarily with no absolute conditions. It can send contradictory messages to other components or simply remain silent. It may look normal from outside.

Consensus is usually associated with replicated state machines, a general approach to building fault-tolerant systems. In a replicated state machine, each instance has a log. A consensus algorithm is used to agree on the commands in the servers' logs. The consensus algorithm ensures that each state machine processes the same series of commands and thus produces the same series of results, arriving at the same series of states.

Note: In addition to the Raft consensus algorithm, to prevent a Byzantine failure GraphDB also stops all write procedures while adding a node to the cluster that is currently being processed. You can learn more about adding nodes to the cluster in the *Manage cluster membership* section of the documentation.

5.1.3.2 High availability deployment

A typical scenario would be a deployment in a cloud infrastructure with the ability to deploy GraphDB instances in different regions or zones so that if a region or zone fails, the GraphDB cluster will continue functioning without any issues for the end user.

To achieve high availability, we recommend deployment of GraphDB instances in different zones or regions while considering the need for a majority *quorum* in order to be able to accept INSERT and DELETE requests. This means that the deployment should always have over 50% of the instances running.

Another recommendation is to distribute the GraphDB instances so that you do not have exactly half of the GraphDB instances in one zone and the other half in another zone, as this way it would be easy to lose the majority quorum. In such cases, it is better to use three zones.

For complex multi-zone deployments with interregional latency, it is best to use the *Multi-region clusters mechanism*.

Cluster group with three nodes

In a cluster with three nodes, you need at least two in order to write data successfully. In this case, the best deployment strategy is to have three GraphDB instances distributed in three zones in the same region. This way, if one zone fails, the other two instances will still form a quorum and the cluster will accept all requests.

Note: Having the instances in different regions may introduce latency.

Cluster group with five nodes

In a cluster with five nodes, you need three nodes for a quorum. If you have three available regions or zones, you can deploy two instances in zone 1, two in zone 2, and one in zone 3.

If any of the zones fail, you would still have at least three more GraphDB instances that will form a quorum.

5.1.4 Quorum-based replication

The GraphDB cluster relies on quorum-based replication, meaning that the cluster should have over 50% alive nodes in order to be able to execute INSERT or DELETE operations. This ensures that there will always be a majority of GraphDB nodes that always have up-to-date data.

If any nodes are unavailable when an INSERT or DELETE operation is executed, but more than 50% of the nodes are alive, the request will be accepted, distributed among the reachable alive nodes, and saved if everything is OK. Once the unavailable nodes come back online, the transactions will be distributed to them as well.

If fewer than 50% of the nodes are available, any INSERT or DELETE operations will be rejected.

5.1.4.1 Query load balancer

For maximum efficiency, the GraphDB cluster distributes the incoming read queries to all nodes, prioritizing the nodes that have fewer running queries. This ensures the optimal hardware resource utilization of all nodes.

5.1.4.2 Local consistency

GraphDB supports two types of local consistency: **None** and **Last Committed**.

- **None** is the default setting and is used when no local consistency is needed. In this mode, queries will be sent to any readable node with no guarantee of strong consistency. This is suitable for cases where eventual consistency is sufficient or when enforcing strong consistency is too costly.
- **Last Committed** is used when strong consistency is required, ensuring that the results reflect the state of the system after all transactions have been committed. This could lead to lower scalability, however, as the set of nodes to which a query could be load-balanced is smaller. In this mode, queries will be sent to a readable node that has advanced to the last transaction.

The choice between **None** and **Last Committed** depends on the specific requirements and constraints of the application and use case. In general, if query results should always reflect the up-to-date state of the database, **Last Committed** should be used. Otherwise, **None** is sufficient.

5.2 Creating and managing a cluster

These instructions walk through the steps for creating and monitoring a *cluster group*.

5.2.1 Prerequisites

- At least three licensed GraphDB installations are available. The Raft algorithm recommends an odd number of nodes, so a cluster of five nodes is a good choice.
- All instances have the same security settings, in particular the same shared Token secret. For each instance, inside the `graphdb.properties` file set the `graphdb.auth.token.secret` property to the desired Token secret value `graphdb.auth.token.secret = <my-shared-secret-key>` where `<my-shared-secret-key>` can be any string.
- All nodes must have networking configured correctly. The hostname provided by the operating system must be resolvable to the correct IP address on each node in the cluster.
- Some of the GraphDB configuration properties need to be shared between all nodes. See below.

> **Tip:** In case your networking is not configured correctly or you are unsure if it is, you can set the hostname for each node by configuring `graphdb.hostname = <your.example.hostname>` in each `graphdb.properties` file on each instance. If you do not have resolvable hostnames, you can supply an IP address instead.

The examples below assume that there are five nodes reachable at the hostnames `graphdb1.example.com`, `graphdb2.example.com`, `graphdb3.example.com`, `graphdb4.example.com`, and `graphdb5.example.com`.

5.2.1.1 Ports

The following table outlines the network connectivity requirements for the GraphDB cluster nodes and optionally, the cluster proxies.

The table lists the default ports used by GraphDB, but can be changed using configuration properties.

Source	Destination	Port/Protocol	Direction	Purpose
Nodes	Nodes	7200/TCP	Bidirectional	Request proxy/Configuration synchronization
Nodes	Nodes	7300/TCP	Bidirectional	Raft communication
Cluster proxy	Nodes	7200/TCP	Incoming	Request proxy
Cluster proxy	Nodes	7300	Bidirectional	Node state observer

5.2.1.2 Configuration properties prerequisites

Some of the GraphDB properties should be identical on all nodes in the cluster — notably, all security-related properties have to be identical on all nodes. The list below lists all configuration properties and configuration property groups that must be identical between the nodes.

- `graphdb.auth.*`
- `graphdb.ontop.*`
- `graphdb.s3.*`
- `graphdb.audit.*`
- `graphdb.cluster.*`
- `graphdb.raft.*`
- `graphdb.engine.*`
- `graphdb.health.*`
- `graphdb.llm.*`
- `graphdb.openai.*`
- `graphdb.ttyg.*`
- `graphdb.gpt-sparql.*`
- `graphdb.request.id.alternatives`
- `graphdb.append.request.id.headers`
- `graphdb.wait.report.minutes`
- `graphdb.jsonld.whitelist`
- `graphdb.heapdump.enable`
- `graphdb.inference.buffer`
- `graphdb.inference.concurrency`
- `graphdb.workbench.cors.enable`
- `graphdb.workbench.cors.origin`
- `graphdb.workbench.cors.expose-headers`
- `graphdb.workbench.maxConnections`
- `graphdb.workbench.maxUploadSize`
- `graphdb.proxy.followerRetries`
- `graphdb.persistent.parallel.inferencers`
- `graphdb.connector.ssl.enabled`
- `graphdb.connector.ssl.certificateKeystorePassword`
- `graphdb.connector.ssl.certificateKeyAlias`
- `graphdb.connector.ssl.certificateKeyPassword`
- `graphdb.connector.ssl.truststorePassword`

5.2.1.3 Achieving a high availability deployment

A typical scenario would be a deployment in a cloud infrastructure with an odd number of GraphDB instances in different regions or zones. No single zone should contain more than 50% of the nodes in the cluster, so that the GraphDB cluster can continue to function even if one of the zones fails. We recommend deploying clusters with either three or five nodes as the optimal option. To learn more about high availability deployment, check the *examples of high availability cluster deployments*.

5.2.2 Create cluster

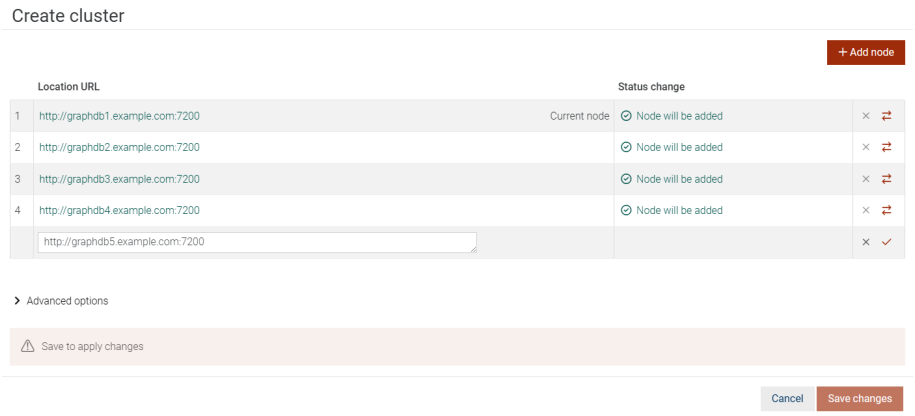
A cluster can be created interactively from the Workbench or programmatically via the REST API.

5.2.2.1 Using the Workbench

1. Open any of the GraphDB instances that you want to be part of the cluster, for example http://graphdb1.example.com:7200, and go to *Setup* ▸ *Cluster*.

   Click the icon to create a cluster group.

2. In the dialog that opens, add all the nodes you want to be part of the cluster. To add a cluster node, click *Add node* and enter the address of the node you want to add, then click on the check icon to confirm it:



Note: The nodes will become a part of the cluster only once you confirm the cluster creation by clicking

on *Save changes*.

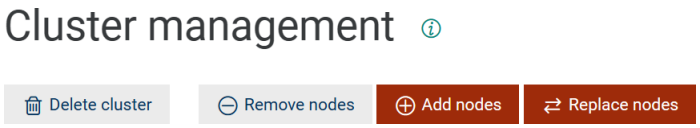
Clicking on *Advanced settings* opens an additional panel with *settings* that affect the entire cluster group. You can also leave the default parameters when creating the cluster, and if needed change them later once the cluster has been created.

3. Click *Save changes* to create the cluster. One of the nodes will become a candidate and initiate an election (you may see it briefly flash in green), and eventually a leader will be elected and shown in dark red.

In this example, `graphdb1.example.com` became the leader but it could have been any of the other four nodes. The fact that `graphdb1.example.com` was used to create the cluster does not affect the leader election process in any way.

Tip: You can open a legend listing all cluster states by clicking on the Question mark on the bottom left of the screen. To learn more about the various cluster states, check the *cluster states* documentation, which explains the legend in further detail.

4. You can also *add, remove or replace nodes from the cluster group*, as well as *delete* it.



See also *Using a cluster*.

5.2.2.2 Using the REST API

You can also create a cluster using the respective REST API — see *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller* for the interactive REST API documentation.

The examples below use the curl command line utility.

To create the cluster group, simply POST the desired cluster configuration to the `/rest/cluster/config` endpoint of any of the nodes (in this case http://graphdb1.example.com:7200):

```
curl -X POST -H 'Content-type: application/json' \
    http://graphdb1.example.com:7200/rest/cluster/config \
    -d '{
        "nodes": [
            "graphdb1.example.com:7300",
            "graphdb2.example.com:7300",
```

(continues on next page)

(continued from previous page)

```
            "graphdb3.example.com:7300"
        ]
    }'
```

Each node uses the default HTTP port of 7200 and the default RPC port of 7300.

Tip: The default RPC port is the HTTP port + 100. Thus, when the HTTP port is 7200, the RPC port will be 7300. You can set a custom RPC port in the `graphdb.properties` file of a given node's GraphDB instance using `graphdb.rpc.port = NNNN`, where `NNNN` is the chosen port.

Just like in the Workbench, you do not need to specify the advanced settings if you want to use the defaults. If needed, you can specify them like this:

```
curl -X POST -H 'Content-type: application/json' \
    http://graphdb1.example.com:7200/rest/cluster/config \
    -d '{
        "nodes": [
            "graphdb1.example.com:7300",
            "graphdb2.example.com:7300",
            "graphdb3.example.com:7300"
        ],
        "electionMinTimeout": 8000,
        "electionRangeTimeout": 6000,
        "heartbeatInterval": 2000,
        "messageSizeKB": 64,
        "transactionLogMaximumSizeGB": 50,
        "verificationTimeout": 300
    }'
```

The HTTP return code for this operation may be 201, 400, or 412:

- **201: Cluster successfully created**

  If the cluster group has been successfully created, you will get a `201` Success response code, and the returned response body will indicate that the cluster is created on all nodes:

  ```
  {
    "graphdb1.example.com:7300": "CREATED",
    "graphdb2.example.com:7300": "CREATED",
    "graphdb3.example.com:7300": "CREATED"
  }
  ```

- **400: Invalid cluster configuration**

  If the JSON description of the cluster group configuration is invalid, the returned response code will be `400` Bad request.

- **412: Unreachable nodes or cluster already existing on a node**

  If during the creation of the cluster some nodes are unreachable or a cluster group already exists on a given node, the response code will be `412` Precondition failed. The error will be shown in the JSON response body:

  – Unreachable node:

  ```
  {
    "graphdb1.example.com:7301": "NOT_CREATED",
    "graphdb2.example.com:7302": "NO_CONNECTION",
    "graphdb3.example.com:7303": "NOT_CREATED"
  }
  ```

  – Cluster already existing on a node:

```
{
  "graphdb1.example.com:7301": "NOT_CREATED",
  "graphdb2.example.com:7302": "ALREADY_EXISTS",
  "graphdb3.example.com:7303": "NOT_CREATED"
}
```

5.2.2.3 Creation parameters

The cluster group configuration has several properties that have typical default values:

Parameter	Default value	Description
`electionMinTimeout`	8000	The minimum wait time in milliseconds for a heartbeat from a *leader*.
`electionRangeTimeout`	6000	The variable portion of each waiting period in milliseconds for a heartbeat.
`heartbeatInterval`	2000	The interval in milliseconds between each heartbeat that is sent to *follower* nodes by the leader.
`verificationTimeout`	300	The amount of time in milliseconds that a follower node would wait before attempting to verify the last committed entry when the first verification is unsuccessful.
`messageSizeKB`	64	The size of the data blocks, in kilobytes, transferred during data replication streaming through the RPC protocol.
`transactionLogMaximumSizeGB`	50	Maximum size of the transaction log in GBs. The transaction log will be automatically truncated if it becomes bigger than this value. The minimum transaction log size is 1 GB. Setting it to a negative value will disable automatic transaction log truncation.
`batchUpdateInterval`	5000	The interval in milliseconds between requesting updates from the primary cluster. Used only when the cluster is in *secondary mode*.

5.2.3 Manage cluster membership

You can add, remove, or replace cluster nodes at runtime without having to stop the entire cluster group. To achieve this, when making a change to the cluster membership, GraphDB checks whether there is total consensus between the nodes in the new configuration.

Tip: A **total consensus** means that all nodes in the configuration have successfully processed the last entry

If you try to join a node that used to be a part of the cluster but has since disconnected from it, only the necessary metadata will be sent to that node, provided the transaction log has not changed since disconnecting. This way bulky snapshot replications are avoided when joining previously in-sync nodes.

If the majority of nodes fail to respond to *heartbeats*, you can remove the non-responsive nodes all at once. In that case, a majority consensus on the new configuration would be enough for this operation to be executed successfully.

Note: To maintain a majority consensus, GraphDB will prevent you from removing 50% or more of the nodes from a current configuration at once.

You can also directly replace a node in the cluster with another instance, even if the address of the node you attempt to add is not part of the cluster.

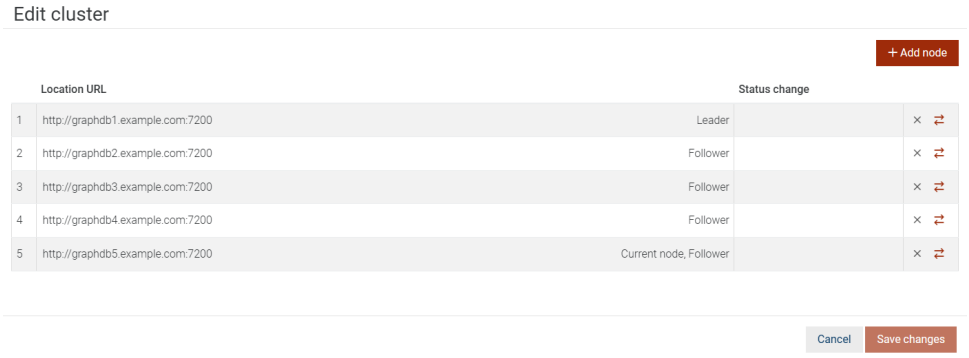
The replace procedure allows you to replace more than one node, or to add a number of new nodes different than the number of old nodes you are removing — as long as you are not removing the majority of nodes and there is majority consensus between the nodes in the new group.

Warning: When adding nodes, GraphDB stops all current write operations in order to prevent a *Byzantine fault issue*. The writes are resumed once the add node procedure has concluded.

5.2.3.1 Managing cluster membership through Workbench

Clicking on the *Edit cluster* button opens up a pop-up interface that allows you to easily add, remove and replace nodes in the current configuration. You can simultaneously perform any combination of those three actions, as long as you preserve the total consensus between the nodes in the new configuration.

The *Edit cluster* interface also displays additional information, such as the status of the individual nodes as well as the current node.



Note: Any changes made to the cluster configuration will not be preserved until you click on *Save changes* to create the cluster.

Add nodes through the Workbench

1. From the cluster view, open the *Edit cluster* interface.
2. Click on *Add node*.
3. Add the URL address of the node you want to add, and confirm by clicking on the check icon.
4. Save the changes to the cluster.

Remove nodes through the Workbench

1. From the cluster view, open the *Edit cluster* interface.
2. Click on the *Remove node* icon next to the node you want to remove.
3. Save the changes to the cluster.

Warning: To preserve the stability of the cluster, you cannot remove half or more of the active nodes at the same time. If you want to remove more than 50% of the nodes, you will have to split the process in two or more steps.

Remove nodes through the Workbench

1. From the cluster view, open the *Edit cluster* interface.
2. Click on the *Replace node* icon next to the node you want to replace.
3. Add the URL address of the node you want to add in its place, and confirm by clicking on the check icon.
4. Save the changes to the cluster.

> **Warning:** To preserve the stability of the cluster, you cannot replace half or more of the active nodes at the same time. If you want to replace more than 50% of the nodes, you will have to split the process in two or more steps.

5.2.3.2 Managing cluster membership using the REST API

You can also use the respective REST API — see *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller* for the interactive REST API documentation.

Add nodes using the REST API

From *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller*, use a *curl* command like the following to send a POST request to the `/rest/cluster/config/node` endpoint:

```
curl -X POST -H 'Content-Type: application/json' \
  'http://graphdb1.example.com:7200/rest/cluster/config/node' \
  -d '{
   "nodes": [
     "graphdb3.example.com:7300"
   ]
 }'
```

If one of the nodes from the group or from the newly added ones has no connection to any of the nodes, an error message will be returned. This is because a total consensus between the nodes in the new group is needed to accept the configuration, which means that all of them should see each other.

If the added node is part of a different cluster, an error message will be returned.

You can make cluster membership changes only if the cluster has a leader node. If you try to add a node to the cluster group while there is no leader, an error message will be returned.

You can add nodes that have previously been part of the same cluster, even if they are not empty. If you add such a node, the cluster will automatically synchronize it. If the added node has not previously been part of the cluster, it should be either empty or in the same state as the cluster, which means that it should have the same repositories and namespaces as the nodes in the cluster. If one of these conditions is not met, you will not be able to add the node.

> **Note:** You can't add a node that was successfully replaced or removed from the cluster back to the cluster if that node is behind. It can be added only if it is completely empty, or if no write actions have been performed on the cluster since the last time that node was a part of the cluster.
>
> If a node was not connected to the cluster when it was removed, but "truncate transaction log" was performed on the cluster, the node cannot be added back to the cluster.

Remove nodes using the REST API

From *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller*, send a DELETE request to the `/rest/cluster/config/node` endpoint:

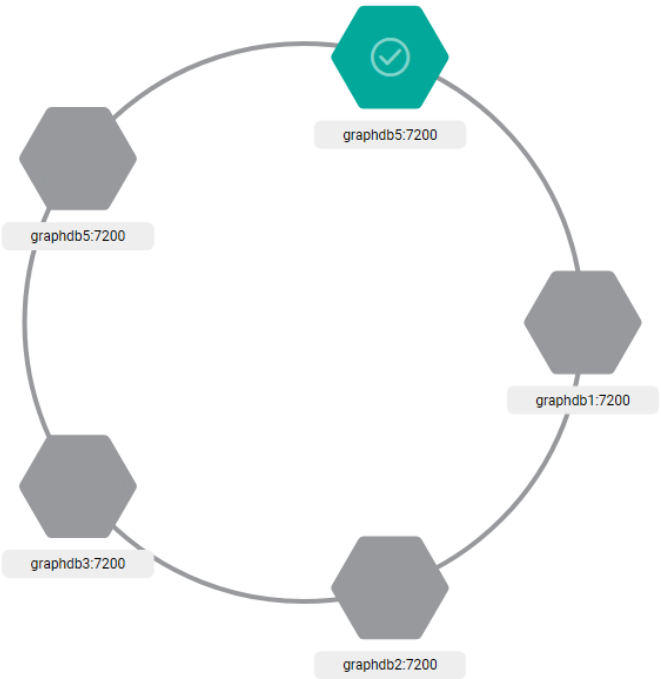
```
curl -X DELETE -H 'Content-Type: application/json' \
'http://graphdb1.example.com:7200/rest/cluster/config/node' \
  -d '{
   "nodes": [
     "graphdb3.example.com:7300"
   ]
 }'
```

If one of the nodes remaining in the new cluster configuration is down or not visible to the others, the operation will not be successful. This is because a total consensus between all nodes in the new configuration is needed, so all of them should see each other.

You can make cluster membership changes only if the cluster has a leader node. If you try to remove a node from the cluster group while there is no leader, an error message will be returned.

You can also remove the leader node — in that case, the cluster will elect a new leader at random.

If a node is down, it can still be removed as it will not be part of the new configuration. If started again, the node will "think" that it is still part of the cluster and will be stuck in candidate state. The rest of the nodes will not accept any communication coming from it. In such a case, you can only manually delete the cluster (using *Show cluster configuration* ▸ *Delete cluster*) from this node.



Replace nodes using the REST API

From *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller*, send a POST request to the `/rest/cluster/config/node` endpoint:

```
curl -X PATCH -H 'Content-Type: application/json' \
  'http://graphdb1.example.com:7200/rest/cluster/config/node' \
  -d '{
   "addNodes": [
     graphdb4.example.com:7300
     graphdb5.example.com:7300
   ],
   "removeNodes": [
     graphdb3.example.com:7300
   ]
 }'
```

A total consensus between the nodes in the resulting group is needed to accept the configuration, which means that all of the nodes in the cluster (including the remaining and the newly added nodes) should see each other. If one of them has no connection to any of the nodes, replace operation will fail, an error message will be returned, and no changes to the current cluster group will be made.

If the added node is part of a different cluster, an error message will be returned.

You can make cluster membership changes only if the cluster has a leader node. If you try to replace nodes in the cluster group while there is no leader, an error message will be returned.

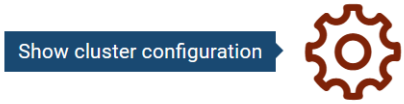
You can also replace the leader node — in that case, the cluster will elect a new leader at random.

5.2.4 Manage cluster configuration properties

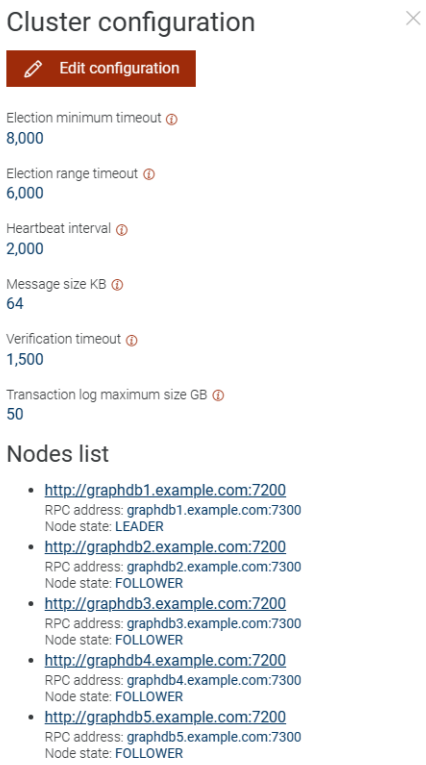
You can view and manage the cluster configuration properties both from the Workbench and the REST API.

5.2.4.1 Using the Workbench

To view the properties, go to *Setup* ▸ *Cluster* and click the cog icon on the top right.



It will open a panel showing the cluster group configuration properties and a list of its nodes.



To modify the configuration properties, click *Edit configuration*.

> **Warning:** Editing of these properties is only possible **on the leader node**.

5.2.4.2 Using the REST API

To view the cluster configuration properties, go to *Help* ▸ *REST API* ▸ *GraphDB Workbench API* ▸ *cluster-group-controller* and perform a GET request to the `/rest/cluster/config` endpoint on any of the nodes:

```
curl http://graphdb1.example.com:7200/rest/cluster/config
```

To check the cluster configuration, send a GET request to the `GET /rest/cluster/config` endpoint. The HTTP return code for this operation may be 200 or 404:

- **200: Returns cluster configuration**

  If the cluster configuration has passed successfully, the response code will be `200` Success.

- **404: Cluster not found**

  If no cluster group has been found, the returned response code will be `404`. One such case could be when you attempt to create a cluster group with just one GraphDB node.

To update the configuration properties, perform a PATCH request containing the parameters of the new configuration to the `/rest/cluster/config` endpoint:

```
curl -X PATCH -H 'Content-Type: application/json' \
  'http://graphdb1.example.com:7200/rest/cluster/config' \
  -d '{
   "electionMinTimeout": 8000,
   "electionRangeTimeout": 6000,
   "heartbeatInterval": 2000,
   "messageSizeKB": 64,
```

(continues on next page)

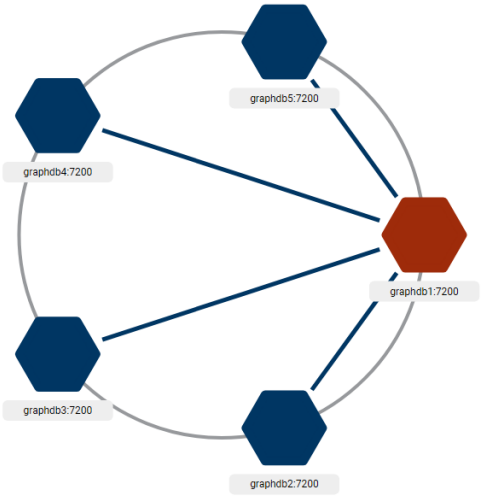
(continued from previous page)

```
  "transactionLogMaximumSizeGB": 50,
  "verificationTimeout": 300
}'
```

If one of the cluster nodes is down or was not able to accept the new configuration, the operation will not be successful. This is a total consensus between the nodes is required. In case one or more of them cannot append the new configuration, it will be rejected by all of the nodes.

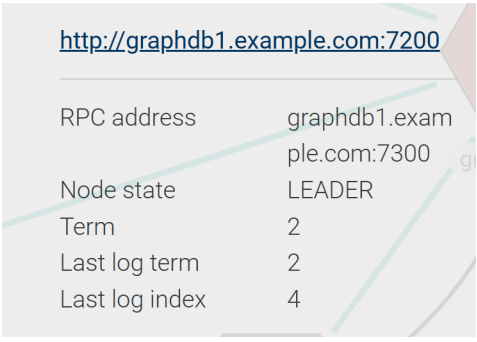
5.2.5 Monitor cluster status

To check the current status of the cluster, including the current leader, open the Workbench and go to *Setup ▸ Cluster*.



The solid green lines indicate that the leader is IN_SYNC with all followers.

Clicking on a node displays some basic information about it, such its state (leader or follower) and RPC address. Clicking on its URL will open the node in a new browser tab.



You can also use the REST API to get more detailed information or to automate monitoring.

5.2.5.1 Cluster group

To check the status of the entire **cluster group**, send a GET request to the `/rest/cluster/group/status` endpoint of any of the nodes, for example:

```
curl http://graphdb1.example.com:7200/rest/cluster/group/status
```

If there are no issues with the cluster group, the returned response code will be `200` with a result similar to the following:

```
[
   {
      "address" : "graphdb1.example.com:7300",
      "endpoint" : "http://graphdb1.example.com:7200",
      "lastLogIndex" : 0,
      "lastLogTerm" : 0,
      "nodeState" : "LEADER",
      "syncStatus" : {
         "graphdb2.example.com:7300" : "IN_SYNC",
         "graphdb3.example.com:7300" : "IN_SYNC"
      },
      "term" : 2,
      "recoveryStatus" : {}
   },
   {
      "address" : "graphdb2.example.com:7300",
      "endpoint" : "http://graphdb2.example.com:7200",
      "lastLogIndex" : 0,
      "lastLogTerm" : 0,
      "nodeState" : "FOLLOWER",
      "syncStatus" : {},
      "term" : 2,
      "recoveryStatus" : {}
   },
   {
      "address" : "graphdb3.example.com:7300",
      "endpoint" : "http://graphdb3.example.com:7200",
      "lastLogIndex" : 0,
      "lastLogTerm" : 0,
      "nodeState" : "FOLLOWER",
      "syncStatus" : {},
      "term" : 2,
      "recoveryStatus" : {}
   }
]
```

Response code `404` will be returned if no cluster has been found.

Note: Any node, regardless of whether it is a leader or a follower, will return the status for all nodes in the cluster group.

A `recoveryStatus` value has the following structure:

```
"recoveryStatus":{
            state:RecoveryStateEnum,
            message:String,
            affectedNodes:[<list of affected node addresses>]
    }
```

The `RecoveryStateEnum` value can be one of the following:

- `SEARCHING_FOR_NODE`

- `WAITING_FOR_SNAPSHOT`
- `RECEIVING_SNAPSHOT`
- `APPLYING_SNAPSHOT`
- `BUILDING_SNAPSHOT`
- `SENDING_SNAPSHOT`

Below are two example values for `recoveryStatus`:

```
"recoveryStatus": {
  "state": "SENDING_SNAPSHOT",
  "message": "Sending a snapshot to node http://graphdb2.example.com:7200",
  "affectedNodes": [
    "http://graphdb2.example.com:7200 "
  ]
}
```

```
"recoveryStatus": {
  "state": "WAITING_FOR_SNAPSHOT",
  "message": "Waiting for snapshot from node http://graphdb1.example.com:7200",
  "affectedNodes": [
    "http://graphdb1.example.com:7200"
  ]
}
```

5.2.5.2 Cluster node

To check the status of a single **cluster node**, send a GET request to the `/rest/cluster/node/status` endpoint of the node, for example:

```
curl http://graphdb1.example.com:7200/rest/cluster/node/status
```

If there are no issues with the node, the returned response code will be `200` with the following information (for a leader):

```
{
   "address" : "graphdb1.example.com:7300",
   "endpoint" : "http://graphdb1.example.com:7200",
   "lastLogIndex" : 0,
   "lastLogTerm" : 0,
   "nodeState" : "LEADER",
   "syncStatus" : {
      "graphdb2.example.com:7300" : "IN_SYNC",
      "graphdb3.example.com:7300" : "IN_SYNC"
   },
   "term" : 2,
   "recoveryStatus" : {}
}
```

For a follower node (in this case `graphdb2.example.com`):

```
{
   "address" : "graphdb2.example.com:7300",
   "endpoint" : "http://graphdb2.example.com:7200",
   "lastLogIndex" : 0,
   "lastLogTerm" : 0,
   "nodeState" : "FOLLOWER",
   "syncStatus" : {},
   "term" : 2,
```

(continues on next page)

(continued from previous page)

```
  "recoveryStatus" : {}
}
```

5.2.6 Delete cluster

To delete a cluster, open the Workbench and go to *Setup* ▸ *Cluster*. Click *Delete cluster* and confirm the operation.

> **Warning:** This operation deletes the cluster group on all nodes, and can be executed from any node regardless of whether it is a leader or a follower. Proceed with caution.

You can also use the REST API to automate the delete operation. Send a DELETE request to the `/rest/cluster/config` endpoint of any of the nodes, for example:

```
curl -X DELETE http://graphdb1.example.com:7200/rest/cluster/config
```

By default, the cluster group cannot be deleted if one or more nodes are unreachable. Reachable here means that the nodes are not in status NO_CONNECTION, therefore there is an RPC connection to them. The HTTP return code for this operation may be 200 or 412:

- **200: Cluster deleted**

  If the deletion is successful, the response code will be `200` and the returned response body:

  ```
  {
    "graphdb1.example.com:7300": "DELETED",
    "graphdb2.example.com:7300": "DELETED",
    "graphdb3.example.com:7300": "DELETED"
  }
  ```

- **412: Unreachable nodes**

  If one or more nodes in the group are not reachable, the delete operation will fail with response code `412` and return:

  ```
  {
    "graphdb1.example.com:7300": "NOT_DELETED",
    "graphdb2.example.com:7300": "NO_CONNECTION",
    "graphdb3.example.com:7300": "NOT_DELETED"
  }
  ```

5.2.6.1 Force parameter

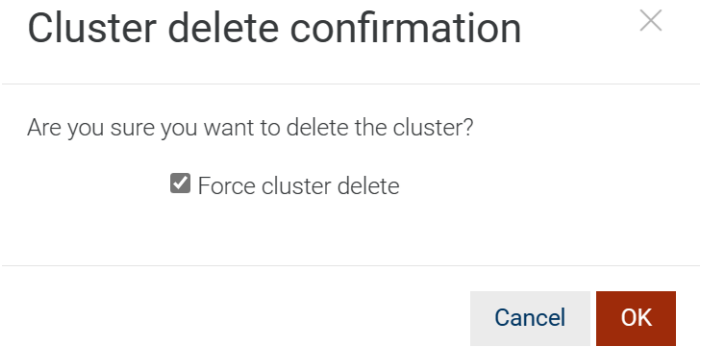
The optional `force` parameter (`false` by default) enables you to bypass this restriction and delete the cluster group on the nodes that are reachable:

- When set to `false`, the cluster configuration will **not be deleted on any** node if **at least one** of the nodes is unreachable.
- When set to `true`, the cluster configuration will be **deleted only on the reachable** nodes, and not be deleted on the unreachable ones.

  In such a case, the returned response will be `200`:

  ```
  {
    "graphdb1.example.com:7300": "DELETED",
    "graphdb2.example.com:7300": "io.grpc.StatusRuntimeException: UNAVAILABLE: io exception",
    "graphdb3.example.com:7300": "DELETED"
  }
  ```

In the Workbench, it can be enabled at deletion:



5.2.7 Configure external cluster proxy

The external cluster proxy can be deployed separately on its own URL. This way, you do not need to know where all cluster nodes are. Instead, there is a single URL that will always point to the leader node.

The externally deployed proxy will behave like a regular GraphDB instance, including opening and using the Workbench. It will always know which node is the current leader and always serve all requests to that leader.

Note: The external proxy does not require a GraphDB SE/EE license.

5.2.7.1 Start the external proxy

To start the external proxy:

1. Execute the `cluster-proxy` script in the `bin` directory of the GraphDB distribution.
2. Provide the cluster secret.
3. Provide a GraphDB server HTTP or RPC address **to at least one of the nodes** in the cluster. You can provide either the HTTP or the RPC address of the node — they are interchangeable. For example:

```
./bin/cluster-proxy -g http://graphdb1.example.com:7200,http://graphdb2.example.com:7200
```

A console message will inform you that GraphDB has been started in proxy mode.

Cluster proxy options

The `cluster-proxy` script supports the following options:

Option	Description
`-d, --daemon`	Daemonize (run in background)
`-r, --follower-retries <num>`	Number of times to retry a request to a different node in the cluster
`-g, --graphdb-hosts <address>`	List of GraphDB nodes' HTTP or RPC addresses that are part of the same cluster
`-h, --help`	Print command line options
`p, --pid-file <file>`	Write PID to <file>
`-Dprop`	Set Java system property
`-Xprop`	Set non-standard Java system property

By default, the proxy will start on port `7200`. To change it, use, for example, `-Dgraphdb.connector.port=7201`.

As mentioned above, the default RPC port of the proxy is the HTTP proxy port + 100, which will be `7300` if you have not used a custom HTTP port. You can change the RPC port by setting, for example, `-Dgraphdb.rpc.port=7301` or `-Dgraphdb.rpc.address=graphdb-proxy.example.com:7301`, e.g.:

```
./bin/cluster-proxy -Dgraphdb.connector.port=7201 -Dgraphdb.rpc.port=7301 -g http://graphdb1.example.
↪com:7200
```

Note: Remember to set the `-Dgraphdb.auth.token.secret=<cluster-secret>` with the same *secret* with which you have set up the cluster. If the secrets do not match, some of the proxy functions may appear as if they are working correctly, but will still be misconfigured and you may experience unexpected behavior at any time.

The external proxy works with two types of cluster node lists: static and dynamic.

- The static list is provided to the proxy through the `-g/--graphdb-hosts` option of the script. This is a comma-separated list of HTTP or RPC addresses of cluster nodes. At least one address to an active node should be provided. Once the proxy is started, it tries to connect to each of the nodes provided in this list. If it succeeds with one of them, it then builds the dynamic cluster node list.
- The dynamic cluster node list is built by requesting the cluster's current status from one of the nodes in the static list. The proxy then subscribes to any changes in the cluster status such as leader changes, nodes being added or removed, or nodes out of reach. The external proxy always sends all requests to the current cluster leader. If there is no leader at the moment, or the leader is unreachable, requests will go to a random node.

Note: The dynamic cluster node list is reset every time the external proxy is restarted. After each restart, the proxy knows only about the nodes listed in the static node list provided by the `-g/--graphdb-hosts` option of the script.

5.2.7.2 External proxy for cluster running over SSL

To set up the external proxy to connect to a cluster over SSL, the same options used to set up GraphDB with security can be provided to the `cluster-proxy` script. The most common ones are:

```
-Dgraphdb.connector.ssl.enabled = true
-Dgraphdb.connector.keystoreFile=<path-to-keystore-file>
-Dgraphdb.connector.keystorePass=<keystore-pass>
-Dgraphdb.connector.keyAlias=<alias-to-key-in-keystore>
```

For more information on the cluster security options, please see below.

5.2.8 Cluster security

5.2.8.1 Encryption

Because there is a lot of traffic between the cluster nodes, it is important that it is encrypted. To do this, make sure that:

- SSL/TLS is enabled on all cluster nodes.
- The nodes' certificates are trusted by the other nodes in the cluster.

The method of enabling SSL/TLS is already described in *Configuring GraphDB instance with SSL*. There are no differences when setting up a node to be used in a cluster.

See how to set up certificate trust between the nodes in the *Certificate trust* section of the *Encryption* page.

5.2.8.2 Access control

Authorization and authentication methods in the cluster do not differ from those for a regular GraphDB instance. The rule of thumb is that all nodes in the cluster group must have the same security configuration.

For example, if *SSL* is enabled on one of the nodes, you must enable it on the other nodes as well. If you have configured *OpenID* on one of the nodes, it needs to be configured on the rest of them as well.

5.2.9 Truncate cluster transaction log

The truncate log operation is used to free up storage space on all cluster nodes by clearing the current transaction log and removing cached recovery snapshots. It can be triggered with a POST request to the `/rest/cluster/truncate-log` endpoint.

Note: The operation requires a healthy cluster — that is, one where a leader node is present and all follower nodes are IN_SYNC. This is because the truncate log operation is propagated to each node in the cluster and truncates the log subsequently on each node through the Raft quorum mechanism.

You can truncate the cluster log with the following curl request:

```
curl -X POST '<base_url>/rest/cluster/truncate-log'
```

5.2.10 Disaster recovery

Sometimes the machine on which a cluster node runs might become unavailable or experience hardware failures. In these cases, manual intervention may be necessary to bring the cluster back to a healthy state.

This section describes how to recover unavailable cluster nodes, or in case of unrecoverable situations, how to rebuild the entire cluster from a single node.

5.2.10.1 Recovering individual nodes

The GraphDB cluster configuration relies heavily on the RPC addresses used for the nodes when the cluster was first created. By default, this is the machine's hostname; however, it can be overridden with the RPC address.

The default setup in most cloud and virtualized environments is for the machines to get a new hostname, IP address, or both.

To recover individual nodes, determine if *quorum* is affected by using the *cluster and node health check endpoints*. If there are not enough nodes to form a quorum, consult the *Rebuilding the cluster from a single node* process.

In case the RPC address has changed:

1. Stop the new GraphDB node from running.
2. Remove the failed node from the cluster.
3. Delete the entire data directory of the new node.
4. Start the new GraphDB node and join it to the cluster.

In case the RPC address **has not** changed, simply start the GraphDB process. The leader should start sending updates to the node; just wait for the node to move into the `IN_SYNC` state.

5.2.10.2 Rebuilding the cluster from a single node

> **Warning:** Make sure you have a backup of the current data on all nodes before continuing with this procedure. Most cloud environments provide the ability to create volume snapshots. If you have enough space on the machine you can move the data instead of deleting it.

Sometimes multiple nodes can fail simultaneously, or nodes might be available but unable to agree on the state of the cluster. When this happens, the cluster might be in a deadlocked state and unable to process write or even read requests.

If this happens, you can rebuild the cluster from a single node serving as the source of truth for the new cluster.

Before starting the recovery process, make sure that the cluster is not currently in the process of recovering:

- Examine the log files — if you see recent messages like "recording snapshot" or "recovering from snapshot" it might mean that the cluster is currently recovering from a snapshot
- Monitor the size of the data directory on the GraphDB nodes — if it is rising, that might also indicate that a snapshot replication is in progress
- Use the cluster and node status endpoints to check if any of the nodes are currently syncing

Also, as mentioned above, make sure you have backed up all current data on all nodes.

1. Stop the GraphDB processes on all nodes
2. Determine which node has processed the most transactions by checking the log index maintained by GraphDB, which can be retrieved for each node by following these steps:

   > **Note:** The node determined in this step will be referred to as **node-x** later on in this procedure.

   1. Get access to the file system where GraphDB stores its data
   2. Determine the size (in bytes) of the `log.index` file. The location of this file is relative to the configured GraphDB data directory and has the following path: `raft/transaction-log/log.index`
   3. Calculate the log index using the following formula: $(size_in_bytes/33) - 1$
   4. Select the node with the highest log index; check the contents and size of the `data/repositories` directory to make sure that it contains all the repositories that you have
3. Delete the entire data directory of GraphDB on all nodes except on **node-x**
4. On **node-x**, delete only the `raft/` subfolder of the data directory
5. Start the GraphDB process again on all nodes
6. Create the cluster again, but make sure to issue the request to the **node-x**

Kubernetes

Since the GraphDB pods cannot be stopped individually to perform the modifications in the data directory, you can do one of the following:

- Change the command of the pods to something that will block indefinitely such as `sleep infinity`; also remember to disable the liveness and readiness probes — otherwise Kubernetes will kill and restart the pods
- Scale the StatefulSet down to zero and mount the volumes somewhere else

5.3 Cluster states and election procedures

This section expands on the topics introduced in *Overview of clusters*.

See also:

- *Using a cluster* provides an overview of using the GraphDB client API to manage your cluster.
- *Creating and managing a cluster* describes how to deploy a cluster in GraphDB using the Workbench or via the REST API.
- *Multi-region cluster deployments* details how to configure multiple clusters in different zones to achieve high availability in multi-region deployments.

The GraphDB cluster is built on the Raft consensus algorithm for managing a replicated log on a distributed state machine. Raft allows a collection of machines to work as a coherent group and survive *failure of some of its members*.

A cluster is made up of two basic node types: **leaders** and **followers**. Usually, it comprises an odd number of nodes in order to tolerate failures.

At any given time, each of the nodes is in one of four states:

- **Leader**: Usually, there is one leader that handles all client requests. If a client contacts a follower, the follower redirects it to the leader.
- **Follower**: A cluster is made up of one leader and all other servers are followers. They are passive, meaning that they issue no requests on their own but simply respond to requests from leaders and candidates.
- **Candidate**: This state is used when electing a new leader.
- **Restricted**: In this state, the node cannot respond to requests from other nodes and cannot participate in election. A node goes into this state when there is a license issue such as an invalid or an expired license.

5.3.1 Leader election

Raft uses a heartbeat mechanism to trigger leader election. When nodes start up, they begin as followers. A node remains in the follower state as long as it is receiving valid remote procedure calls (RPCs) from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time known as the election timeout, then it assumes there is no viable leader and begins an election to choose a new leader. A candidate wins an election if it receives votes from a majority of the servers in the full cluster for that term. Each node can vote for at most one candidate in a given term, on a first-come-first-served basis.

If the cluster gets into a situation where only one node is left, that node will switch to read-only mode. The state shown in its cluster status will switch to candidate, as it cannot achieve a quorum with other cluster nodes when new data is written.

The leader election process follows these steps:

1. After the initial configuration request has been sent, a random node will be set as the leader.
2. If the current leader node stops working for some reason, a new election is held to promote the most voted follower nodes to candidate status, and one of those candidates will become the new leader.
3. The leader node sends a constant heartbeat (a form of node status check to see if the node is present and able to perform its tasks).
4. If only one node is left active for some reason, its status will change to candidate. It will switch to read-only mode to prevent further tinkering with it until more nodes appear in the cluster group.

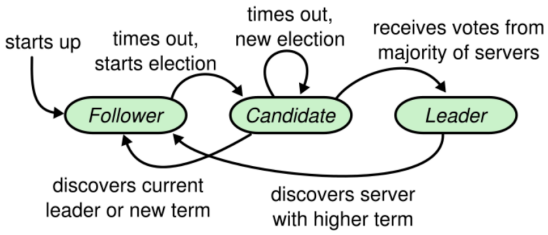


Image source: Stanford Digital Repository

5.3.2 Terms

The Raft algorithm divides time into **terms** of arbitrary length. Terms are numbered with consecutive integers. Each term begins with an election in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the term.

In some situations an election results in a split vote. In this case the term will end with no leader; a new term (with a new election) will commence. Raft ensures that there is at most one leader in a given term.

Different servers may observe the transitions between terms at different times. Raft terms act as a logical clock in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a current term number, which increments with term passings. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to the follower state. If a server receives a request with a stale term number, it rejects the request.

5.3.3 Cluster states

All possible node, sync and connection states are displayed on the bottom left of the *Cluster view* inside Workbench. You can toggle the Cluster states legend by clicking the question mark icon.

5.3.3.1 Cluster node states

Legend	Name	Description
	Leader	The server node that manages the handling of all client requests, including delegation of jobs to the followers.
	Follower	A passive server that doesn't issue requests on its own but simply responds to requests from leaders and candidates — typically, read requests.
	Out of sync	A node that is not in sync and should be recovered.

5.3.3.2 Cluster sync states

Legend	Name	Description
	Candidate	A node that is trying to become a new leader.
	No cluster	A node that is not part of the cluster.
	Out of sync	A node that is not in sync and should be recovered.
	No connection	A node that has no connection to the server.
	Read only	A node that is read only and cannot be written to. A read only node can only be a follower and cannot become a leader.
	Restricted	A node that doesn't have the *required license* to be able to become a part of the cluster.
	Searching for node	An out of sync node that is searching for another node that can provide a snapshot to recover from.
	Waiting for snapshot	An out of sync node that is waiting to receive a snapshot from another node.
	Building a snapshot	A node that is preparing a snapshot, either as part of a recovery procedure or as a scheduled action.
	Sending a snapshot	A node that is currently sending data to recover from.
	Receiving a snapshot	A node that is currently receiving data to recover from.
	Applying a snapshot	A cluster that is applying a received snapshot as part of the recovery procedure.

5.3.3.3 Cluster link states

Legend	Name	Description
	In sync	Displayed between a Leader and a Follower node.
	Syncing	Displayed between the Leader and a node that is currently receiving entries from the Leader.
	Out of sync	Displayed between the Leader node and a node that's out of sync.
	Receiving a snapshot	Shows the direction of a snapshot being sent from one node to another.

5.3.4 Transaction fingerprints

Nodes use **fingerprints** — checksums used to determine whether two repositories are in the same condition and contain the same data. Every transaction performed on a repository returns a fingerprint, which is then compared against the fingerprint of the same repository on the leader node.

When fingerprints do not match, GraphDB automatically resolves the issue by replicating the offending nodes.

5.3.5 Log replication

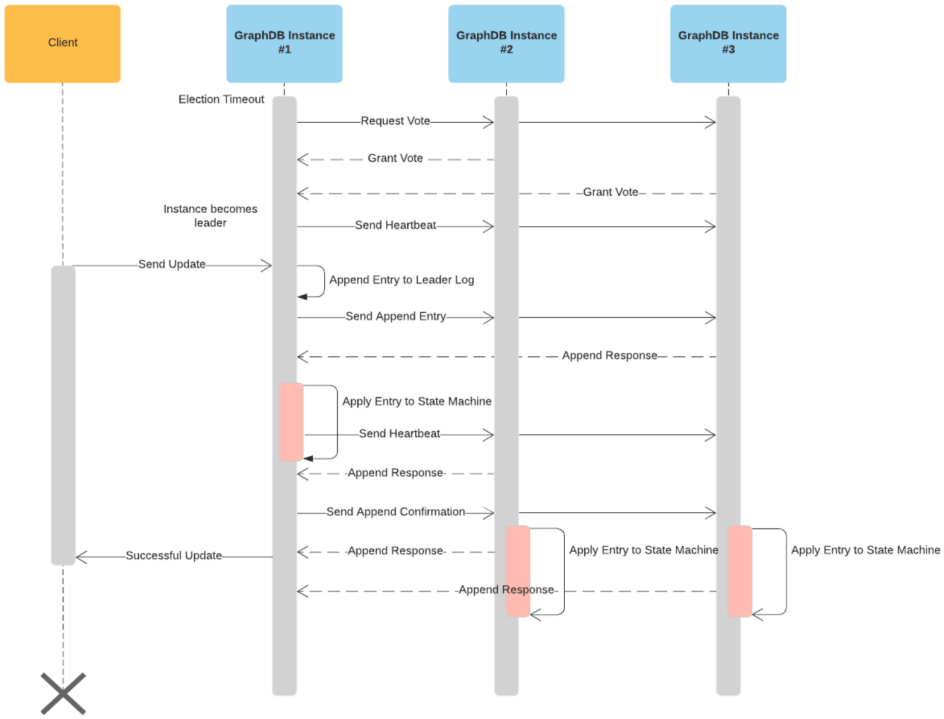
The GraphDB cluster nodes communicate using **remote procedure calls (RPCs)**. The basic consensus algorithm requires only two types of RPCs:

- RequestVote: RPCs that are initiated by candidates during elections
- AppendEntries: RPCs that are initiated by leaders to replicate log entries and to provide a form of heartbeat

Servers retry RPCs if they do not receive a response in a timely manner. They issue RPCs in parallel for best performance.

The log replication resembles a two-phase commit where:

1. The user sends a commit transaction request.
2. The transaction is replicated in the local transaction log.
3. The transaction is replicated to other followers in parallel.
4. The leader waits until enough members (total/2 + 1) have replicated the entry.
5. The leader starts applying the entry to GraphDB successfully.
6. The leader sends a heartbeat until there is a successful commit in GraphDB.
7. The leader sends a second RPC informing followers to apply the log entry to GraphDB.
8. The leader informs the client that the transaction is successful.



Note: If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely in parallel — even after it has responded to the client — until all followers eventually store all log entries.

Only updates relating to repositories, data manipulation, and access are replicated between logs. This includes addition and deletion of repositories, user permission changes, SQL views, smart updates, and standard repository updates.

5.4 Using a cluster

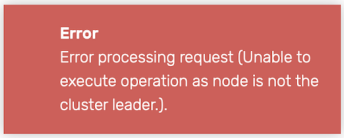
See also:

- *Overview of clusters* provides a brief introduction to clusters in GraphDB.
- *Creating and managing a cluster* walks through the steps for creating and monitoring a cluster group.
- To learn more about clusters, check the *Cluster states and election procedures* section.

Note: This section assumes that you have set up a working cluster.

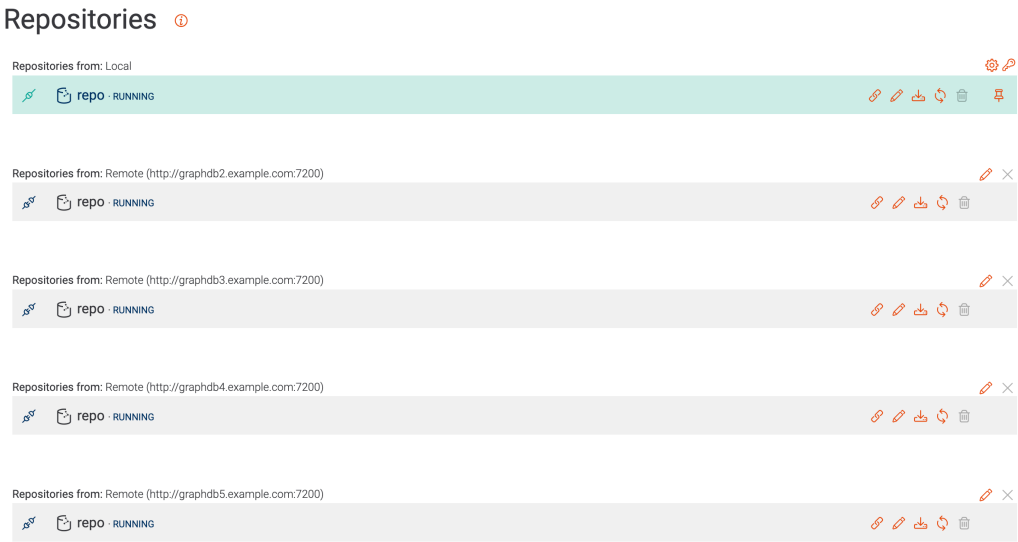
Once *created*, a cluster can be used almost like a single GraphDB configuration. However, all write operations must be performed on the current leader node. Read operations are allowed on any node.

When using the Workbench, make sure you have opened the leader node (go to *Setup* ▸ *Cluster* to check). If you are connected to a follower and try to perform a write operation, you will get an error message:

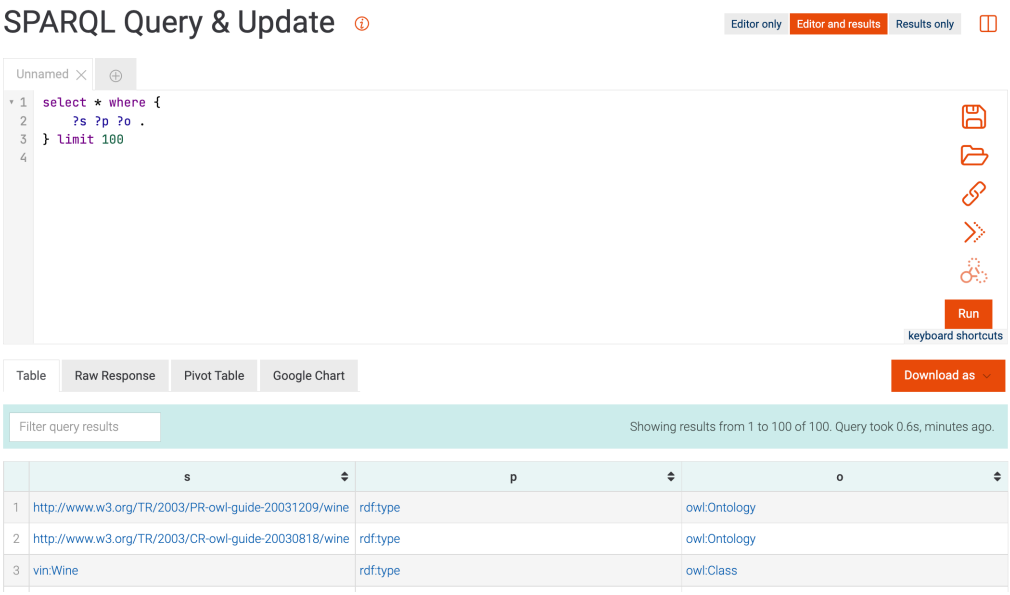


Let's see how some imported data looks in a cluster.

1. On the leader node, in our case `http://graphdb1.example.com:7200`, go to *Setup* ▸ *Repositories* and create a repository (for *Location*, select Local).
2. Note that the repository was created on all nodes because they are connected in a cluster. Unlike GraphDB version 9.x and older where the cluster was defined at repository level, since GraphDB 10, it is defined at instance level. This means that when you create a repository on any of the nodes, it is automatically included in the cluster.
3. Connect to the repository.



4. Import some data in it from *Import* ▸ *User data* ▸ *Upload RDF files*. For this example, let's use the W3.org `wine ontology`.
5. If we open the SPARQL editor and run a basic SELECT query against the imported data, we will see that it behaves just like a regular GraphDB instance.



5.4.1 Using the GraphDB client API for Java

The GraphDB client API for Java is an extension of RDF4J's `HTTPRepository` class that adds support for automatic leader discovery.

The following demonstrates how to create an instance of `GraphDBHTTPRepository`:

```
package com.ontotext.example;

import com.ontotext.graphdb.repository.http.GraphDBHTTPRepository;
import com.ontotext.graphdb.repository.http.GraphDBHTTPRepositoryBuilder;
import org.eclipse.rdf4j.query.TupleQueryResult;
import org.eclipse.rdf4j.repository.RepositoryConnection;

public class Example {
    public static void main(String[] args) {
        GraphDBHTTPRepository repository = new GraphDBHTTPRepositoryBuilder().withRepositoryId("my-repo
↪")
                .withServerUrl("http://graphdb1.example.com:7200")
                .withServerUrl("http://graphdb2.example.com:7200")
                .withCluster()
                .build();

        try (RepositoryConnection connection = repository.getConnection()) {
            connection.begin();
            connection.clear();
            connection.prepareUpdate("insert data { <urn:fact1> a <urn:Fact> ;" +
                            "  <urn:contents> 'GraphDB rocks!'@en }").execute();
            connection.commit();

            String query = "select ?fact ?contents { ?fact a <urn:Fact> ; <urn:contents> ?contents }";
            try (TupleQueryResult tqr = connection.prepareTupleQuery(query).evaluate()) {
                while (tqr.hasNext()) {
                    System.out.println(tqr.next());
                }
            }
        }
    }
}
```

Tip: The client must be configured with at least one server URL that is part of the cluster. The remaining server URLs will be discovered automatically. The server URLs that are provided when the client is created will always be tried, so we recommend that you specify at least two of them in case one of them is down.

Since release 10, GraphDB 10 has included an additional mechanism that allows the use of any cluster nodes with any standard client — for example, RDF4J's `HTTPRepository` or your own software that already works with GraphDB.

The `GraphDBHTTPRepository` class is part of the `graphdb-client-api` module. Use the following Maven configuration to include it in your project:

```
<dependency>
    <groupId>com.ontotext.graphdb</groupId>
    <artifactId>graphdb-client-api</artifactId>
    <version>${graphdb.version}</version>
</dependency>
```

Note: Do not forget to set the `graphdb.version` property to the actual GraphDB version you want to use, or replace the `${graphdb.version}` string with the version.

5.4.2 Using a cluster with an external proxy

The cluster can also be used through an external proxy. To do this, instead of providing the GraphDB HTTP address, provide that of the proxy. For example, if for the repository "myrepo" GraphDB is on `http://graphdb.example.com:7200/repositories/myrepo`, the external proxy will be on `http://graphdb.example.com:7204/repositories/myrepo`.

See how to *configure the external cluster proxy*.

5.4.3 Setting local consistency

Local consistency determines the freshness of the query results. At the lowest level (using the REST API), it is controlled by setting the `X-GraphDB-Local-Consistency` header to one of the following values:

- `last-committed`: Sets Last Committed local consistency. The queries will always return results that include the last completed transaction.
- `none`: Sets no local consistency. The queries may return results from a node that has not yet seen the last completed transaction. This is the default setting.

You can set the header just like any other header in your HTTP client library. For example, with curl:

```
curl 'http://graphdb1.example.com:7200/repositories/myrepo'\
  -H 'X-GraphDB-Local-Consistency: last-committed'\
  -H 'Content-Type: application/sparql-query'\
  -d 'select * { ?s ?p ?o } limit 5'
```

5.4.3.1 Using the GraphDB Java client API

The GraphDB client API for Java has built-in support for setting the local consistency using the `RequestHeaderAware` interface:

```
/**
 * An interface for adding and setting HTTP request headers.
 */
public interface RequestHeaderAware {
    ...
    /**
     * Convenience method for setting the X-GraphDB-Local-Consistency header.
     *
     * @param localConsistency the desired local consistency level
     */
    default void setLocalConsistencyHeader(LocalConsistency localConsistency) {
        setHeader(GraphDBHTTPProtocol.LOCAL_CONSISTENCY_HEADER_NAME, localConsistency.toString());
    }
}
```

The connections returned by `GraphDBHTTPRepository` implement the `RequestHeaderAware` interface. In order to request **Last Committed** local consistency, you need to call `setLocalConsistencyHeader()` with `LocalConsistency.LAST_COMMITTED`:

```
package com.ontotext.example;

import com.ontotext.graphdb.replicationcluster.LocalConsistency;
import com.ontotext.graphdb.repository.http.GraphDBHTTPRepository;
import com.ontotext.graphdb.repository.http.GraphDBHTTPRepositoryBuilder;
import com.ontotext.graphdb.repository.http.RequestHeaderAware;

public class Example {
    public static void main(String[] args) {
        try (RepositoryConnection connection = repository.getConnection()) {
            // Sets local consistency to "Last Committed"
            ((RequestHeaderAware) connection).setLocalConsistencyHeader(LocalConsistency.LAST_
↪COMMITTED);

            // Use connection to evaluate queries
            ...
        }
    }
}
```

5.5 Multi-region cluster deployments

Single-zone and region cluster deployments are great at providing the necessary redundancy for addressing failures of a limited number of instances. However, when dealing with deployments that span multiple regions or zones, interregional latency can add a significant overhead on client queries. The failure of a region or data center (for example, due to a power outage) can also force the cluster to lose its write and read availability if the majority — or all — of the nodes are deployed in the problematic region.

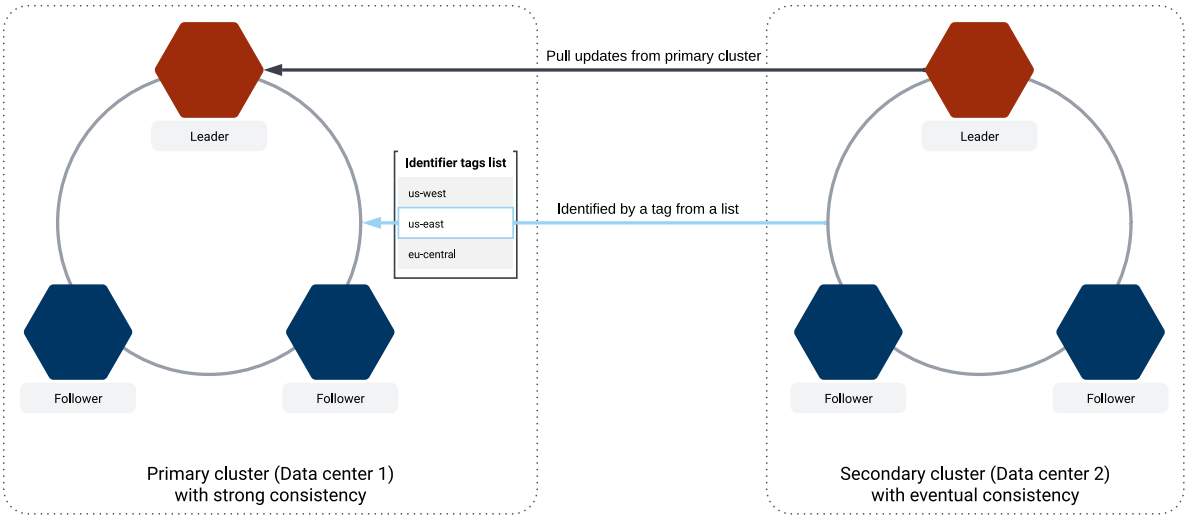
To address these issues and achieve high availability in multi-region deployments, GraphDB has introduced a new cluster type to the Raft protocol called a **secondary cluster**.

5.5.1 Architecture

Secondary cluster is a read-only cluster type that replicates the state of a primary cluster and achieves eventual consistency, even in configurations with significant latency. Similar to the observer nodes, secondary clusters cannot process new updates — they can only keep up to date with the primary cluster by pulling batched updates over a given time interval. The duration of the interval is controlled individually for each secondary cluster by the `batchUpdateInterval` *cluster configuration parameter*.

Tip: A primary cluster is any GraphDB cluster that is not set to a read-only secondary cluster mode, including any cluster in a single-region deployment. However, to distinguish between single- and multiple-region deployments, the term *primary cluster* is almost exclusively used only within the context of multi-region deployments.

Primary and secondary clusters work in harmony to reduce both recovery time and recovery point objectives (RTO and RPO) for multi-region and data center cluster deployments. For any given multi-region deployment, only one primary cluster can exist, and it is responsible for processing all updates. The primary cluster must be healthy (in other words, have a majority quorum) to be able to process updates. The primary cluster will always have the most up-to-date information while secondary clusters will always lag behind (unless the dataset is static) due to the asynchronous nature of the pulling update mechanism.



Note: The example diagram above shows an example configuration of a single primary and a single secondary cluster. However, a multi-region deployment can include any number of secondary clusters — but only a single primary cluster, no matter how secondary ones it has. They can also include a different number of nodes.

Both the primary and secondary clusters operate independently of each other, each with its own separate topology — including a separate leader — as well as all other fault tolerance features available in a single region cluster. Each individual cluster must elect its own leader from the local cluster topology; this leader is responsible for propagating updates and verifying quorum consistency.

Warning: The secondary cluster cannot be created to increase the availability of a primary cluster that is already down. The mechanism exists to provide high availability for writes only in an already established cluster topology.

While the secondary cluster replicates the state of the primary cluster, the primary and the secondary cluster maintain separate log indexes. For example, if a secondary cluster's own internal last log index is 8, that does not respond to the log index 8 of the primary cluster. It also does not mean that the last log index received pulled from the primary cluster is log index 8.

5.5.2 Secondary cluster identifier tags

One of the main components of the secondary cluster mechanism is the use of **identifier tags** — labels used to identify individual secondary clusters by geographic zone, data center, or other unique attribute. For example, you can use a `us-east` tag to identify a secondary cluster located on the East coast of the Unites States.

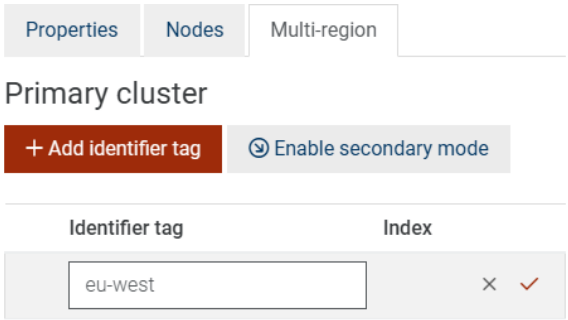
Tags are added to a list that is part of the primary cluster's configuration and declared by the secondary cluster when switching to a secondary mode. The primary cluster does not maintain a list of secondary clusters — only a list of tags associated with each of those clusters.

Identifier tags can take any string value as long as it is not empty. A unique identifier tag must be used for each individual secondary cluster (in other words, you cannot use the same tag to identify two different servers). Secondary clusters cannot be identified by more than one tag.

> **Warning:** A tag must be added to the list maintained by the primary cluster before you enable the cluster's secondary mode. Attempting to enable the secondary mode before adding that cluster's identifier tag to the primary cluster's tags list will result in an error.

5.5.2.1 Adding identifier tags using Workbench

From the *Cluster view* interface, click the gear icon and go to the *Multi-region* ‣ *Add identifier tag*.



Once added, the tag will appear in the list of identifier tags. The *Index* column shows the last index received by the secondary cluster identified by the tag.

> **Tip:** Until a cluster has declared itself as a secondary cluster using the identifier tag you just added, that value will remain zero. You can find more information below in the section on *monitoring multi-region cluster deployments*.

Remove tags from the list by clicking the trash bin icon.

5.5.2.2 Adding identifier tags using the REST API

You can also add identifier tags using the REST API — see *Help* ‣ *REST API* ‣ *GraphDB Workbench API* ‣ *cluster-group-controller* for the interactive REST API documentation. The examples below use the curl command line utility.

To add an identifier tag to the list of tags maintained by the primary cluster, POST the that tag to the `/rest/cluster/config/tag` endpoint of any of the primary cluster's nodes. The example below adds the identifier tag `us-west` to the primary cluster through the node http://graphdb1.example.com:7200.

```
curl -X POST -H 'Content-type: application/json' \
    http://graphdb1.example.com:7200/rest/cluster/config/tag \
    -d '{
        "tag": "us-west"
    }'
```

You can also check which tags have been added to the list by performing a GET request to `/rest/cluster/group/status`.

```
curl -X GET --header http://graphdb1.example.com:7200/rest/cluster/group/status
```

You can delete identifier tags from the list by using the DELETE method with the `/rest/cluster/config/tag` endpoint.

```
curl -X DELETE -H 'Content-type: application/json' \
    http://graphdb1.example.com:7200/rest/cluster/config/tag \
    -d '{
        "tag": "us-west"
    }'
```

Tip: You can remove identifier tags regardless of whether a secondary cluster using that tag is already connected to the primary cluster. This will not result in any failures on either cluster, but that secondary cluster will no longer replicate the state of the primary cluster until the identifier tag is added to the list once again.

Note: Only one identifier tag can be added to or deleted from the list at a time.

5.5.3 Enable the secondary cluster mode

You can switch any healthy primary cluster to secondary mode by providing the RPC address of the primary cluster node that it will replicate and then declaring the tag used to identify it. Once you have switched a cluster to secondary mode, all of its current repositories and data will be removed and replaced with the data and state of the primary cluster.

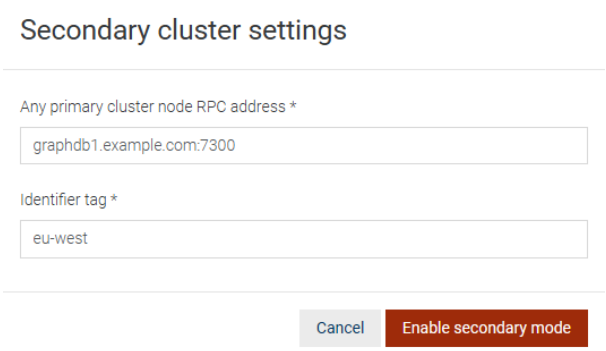
Tip: You can enable the secondary cluster mode from any of the nodes of a cluster, not just the leader node. You can also use any of the healthy nodes of the primary cluster to connect to that cluster.

5.5.3.1 Prerequisites

- Both clusters must share *the same cluster secret*.
- *The tags necessary for identifying the secondary cluster* must already be added to the primary cluster's list of tags.

5.5.3.2 Enable the secondary cluster mode using Workbench

From the *Cluster view* interface, click the gear icon and go to the *Multi-region* ▸ *Enable secondary mode*. Then provide the RPC address of a node from the primary cluster, and a tag to identify the secondary cluster.



Warning: Enabling the secondary mode will delete all data on the secondary cluster and replicate the state of the primary cluster.

Once in secondary mode, you can disable secondary mode and then you will no longer be in that mode. Doing this will stop the pulling of updates from the primary cluster.

5.5.3.3 Enable the secondary cluster mode using the REST API

You can enable the secondary mode using the REST API. See *Help ▸ REST API ▸ GraphDB Workbench API ▸ cluster-group-controller* for the interactive REST API documentation. The examples below use curl.

To enable the secondary mode of a cluster, perform a POST with that tag to the `/rest/cluster/config/secondary-mode` endpoint of any of the nodes of that cluster. You must provide the RPC address of a primary cluster node and declare the identifier tag of the secondary cluster. The example below enables the secondary mode on a cluster that contains the node http://graphdb-eu-west-1.example.com:7200, connects to the primary cluster through the node with RPC address `graphdb1.example.com:7300`, and identifies the secondary cluster with the identifier tag `eu-west`.

```
curl -X POST -H 'Content-type: application/json' \
    http://graphdb-eu-west-1.example.com:7200/rest/cluster/config/secondary-mode \
    -d '{
        "primaryNode": "graphdb1.example.com:7300",
        "tag": "eu-west"
    }'
```

Warning: Enabling the secondary mode will delete all data on the secondary cluster and replicate the state of the primary cluster.

5.5.4 Monitoring multi-region cluster deployments

Once a multi-region cluster has been configured, you can monitor the deployment using Workbench or the REST API.

You can monitor a multi-region cluster deployment from both the primary and the secondary cluster, but there are differences in what information about the multi-region deployment can be monitored between the two.

	Monitoring from the primary cluster	Monitoring from the secondary cluster
Last log index pulled in a batched update by the secondary cluster	✓	✓ (but only by that specific secondary cluster)
Current latest log index of the primary cluster	✓	×
RPC address of the primary cluster's leader	✓	✓
RPC address of the secondary cluster's leader	×	✓
Full list of identifier tags	✓	×

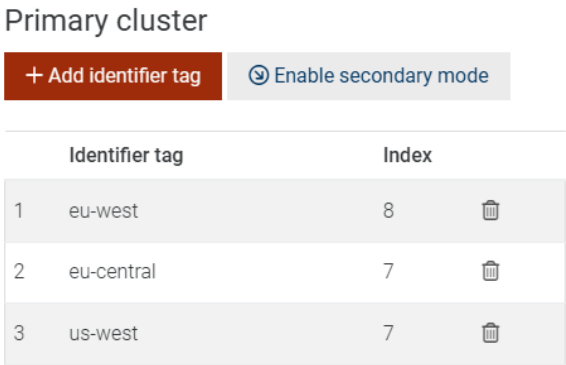
Note: The primary and secondary cluster maintain different log indexes. When monitoring a multi-region deployment, it is important not to confuse the two log indexes and terms.

Tip: Keep in mind that the primary cluster doesn't maintain a list of secondary clusters, but only a list of identifier tags. There is no way to check whether any actual secondary clusters are using those tags at the moment.

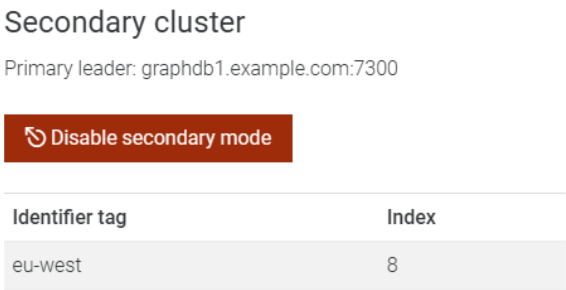
5.5.4.1 Monitoring a multi-region cluster deployment using Workbench

From the Workbench, navigate to *Show cluster configuration* ‣ *Multi-region*.

When viewed from the primary cluster, the tab will display the list of identifier tags and the last log index that each of these tags has requested.



When viewed from the secondary cluster, the tab will display the RPC address of the node of the primary cluster to which the secondary cluster is connected. It will also display the identifier tag associated with the secondary cluster, as well as the last log index received by the secondary cluster from the perspective of the primary cluster.



5.5.4.2 Monitoring a multi-region cluster deployment using the REST API

You can use the REST API to monitor clusters. See *Help ▸ REST API ▸ GraphDB Workbench API ▸ cluster-group-controller* for the interactive REST API documentation. The examples below use curl.

Monitoring a multi-region cluster deployment from the primary cluster

A GET request to the `rest/cluster/group/status` endpoint retrieves information about all of the cluster's nodes as well as information about the list of identifier tags used by the multi-region deployment.

```
curl -X GET --header http://graphdb1.example.com:7200/rest/cluster/group/status
```

Information about the secondary cluster is returned inside `topologyStatus`. Since the primary cluster does not maintain a list of secondary clusters, monitoring the multi-region deployment from the primary cluster provides only a list of associated identifier tags. For each of those tags, the response body provides the last log index update.

In the example below, we see that the primary cluster's last log index is 10. Under `topologyStatus` there is one associated identifier tag `eu-west`, with 8 being the last log index that was sent to it.

```
{
  "address": "graphdb1.example.com:7300",
  "nodeState": "LEADER",
  "term": 2,
  "syncStatus": {
    "graphdb2.example.com:7300": "IN_SYNC",
    "graphdb3.example.com:7300": "IN_SYNC"
  },
  "lastLogTerm": 2,
  "lastLogIndex": 10,
  "endpoint": "http://graphdb1.example.com:7200",
  "recoveryStatus": {},
  "topologyStatus": {
    "state": "PRIMARY_NODE",
    "primaryTags": {
      "eu-west": 8
    },
{
  "address": "graphdb2.example.com:7300",
  "nodeState": "FOLLOWER",
  "term": 2,
  "syncStatus": {},
  "lastLogTerm": 2,
  "lastLogIndex": 10,
  "endpoint": "http://graphdb2.example.com:7200",
  "recoveryStatus": {},
  "topologyStatus": {
    "state": "PRIMARY_NODE"
  }
},
{
  "address": "graphdb3.example.com:7300",
  "nodeState": "FOLLOWER",
  "term": 2,
  "syncStatus": {},
  "lastLogTerm": 2,
  "lastLogIndex": 10,
  "endpoint": "http://graphdb3.example.com:7200",
  "recoveryStatus": {},
  "topologyStatus": {
    "state": "PRIMARY_NODE"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Monitoring a multi-region cluster deployment from the secondary cluster

A GET request to the `rest/cluster/group/status` endpoint retrieves information about all of the cluster's nodes as well as and information about the primary cluster.

```
curl -X GET --header http://graphdb-eu-west-1.example.com:7200/rest/cluster/group/status
```

The response body includes the topology of the cluster, the address of the primary cluster's leader, and the last log index received from the primary cluster.

Note: Monitoring from a secondary cluster cannot return any information about any other secondary clusters connected to the primary cluster or the full list of identifier tags added to the primary cluster.

CHAPTER

SIX

ADMINISTRATION

6.1 Setting up your license

GraphDB is offered in two different editions: Free and Enterprise Edition (EE). Standard Edition (SE) is no longer offered but receives legacy support.

All versions of GraphDB require you to provide the appropriate license file before you can use your copy of GraphDB. When installing GraphDB, the license file can be set through the GraphDB Workbench or programmatically.

To receive a copy of the Free license, request your copy of GraphDB Free from the download page. You will receive the Free License on your email address.

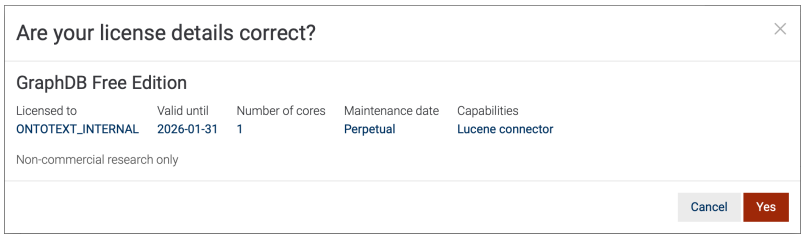
To purchase a license or obtain a copy of EE for evaluation, please contact graphdb-info@ontotext.com.

6.1.1 Setting up your license through the Workbench

1. Add, view, or update your license from *Setup* ‣ *License* ‣ *Set new license*.
2. Select *Click here to upload your license file*.

   Tip: You can also copy and paste it in the text area. If you do, you will also need to click on *Register* after pasting your text in the text box.

3. Check whether your license details are correct, and click on *Yes* to confirm.



After you complete these steps, clicking the *License settings* button at the bottom of the GraphDB home page will display your license details.

6.1.2 Setting up your license through a configuration file

GraphDB will look for a `graphdb.license` file in the GraphDB `work` directory (where non-user-definable configurations are stored) under *GraphDB-HOME*. To install a license file there, copy the license file as `graphdb.license`.

6.1.2.1 Custom file path property

You can use the configuration property `graphdb.license.file` to provide a custom path for the license file, for example:

```
graphdb.license.file = /opt/graphdb/my-graphdb-dev.license
```

The license file must be readable by the user running GraphDB.

Note: If you set the license through a file in the `work` directory or a custom path, you will not be able to change the license through the GraphDB Workbench.

6.1.3 Where GraphDB looks for a license

When looking for a license, GraphDB will use the first license it finds in this order:

- The custom license file property `graphdb.license.file`;
- The `graphdb.license` file in the `work` directory;
- A license set through the GraphDB Workbench.

6.2 Security

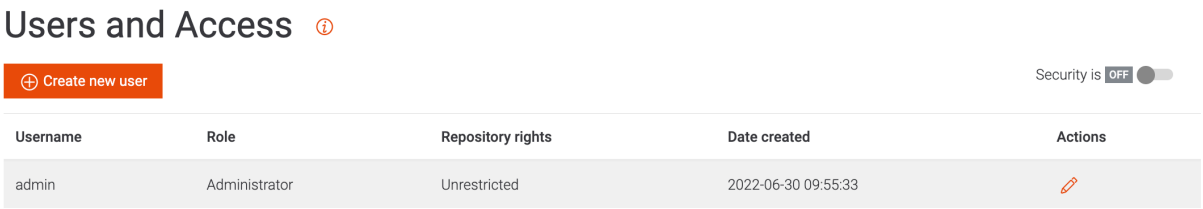
Database security refers to the collective measures used to protect and secure a database from illegitimate use and malicious threats and attacks. It covers and enforces security in several aspects:

6.2.1 Enabling security

Security configurations in the GraphDB Workbench are located under *Setup* ▸ *Users and Access*.

The *Users and Access* page allows you to create new users, edit the profiles, change their password and read/write permissions for each repository, as well as delete them.

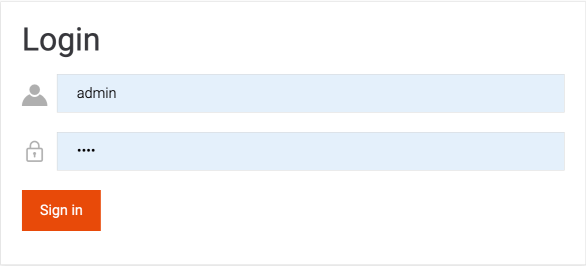
Note: As a security precaution, you cannot delete or rename the admin user.



By default, the security for the entire Workbench instance is disabled. This means that everyone has full access to the repositories and the admin functionality.

To enable security, click the *Security* slider on the top right. You are immediately taken to the login screen.

6.2.1.1 Login and default credentials



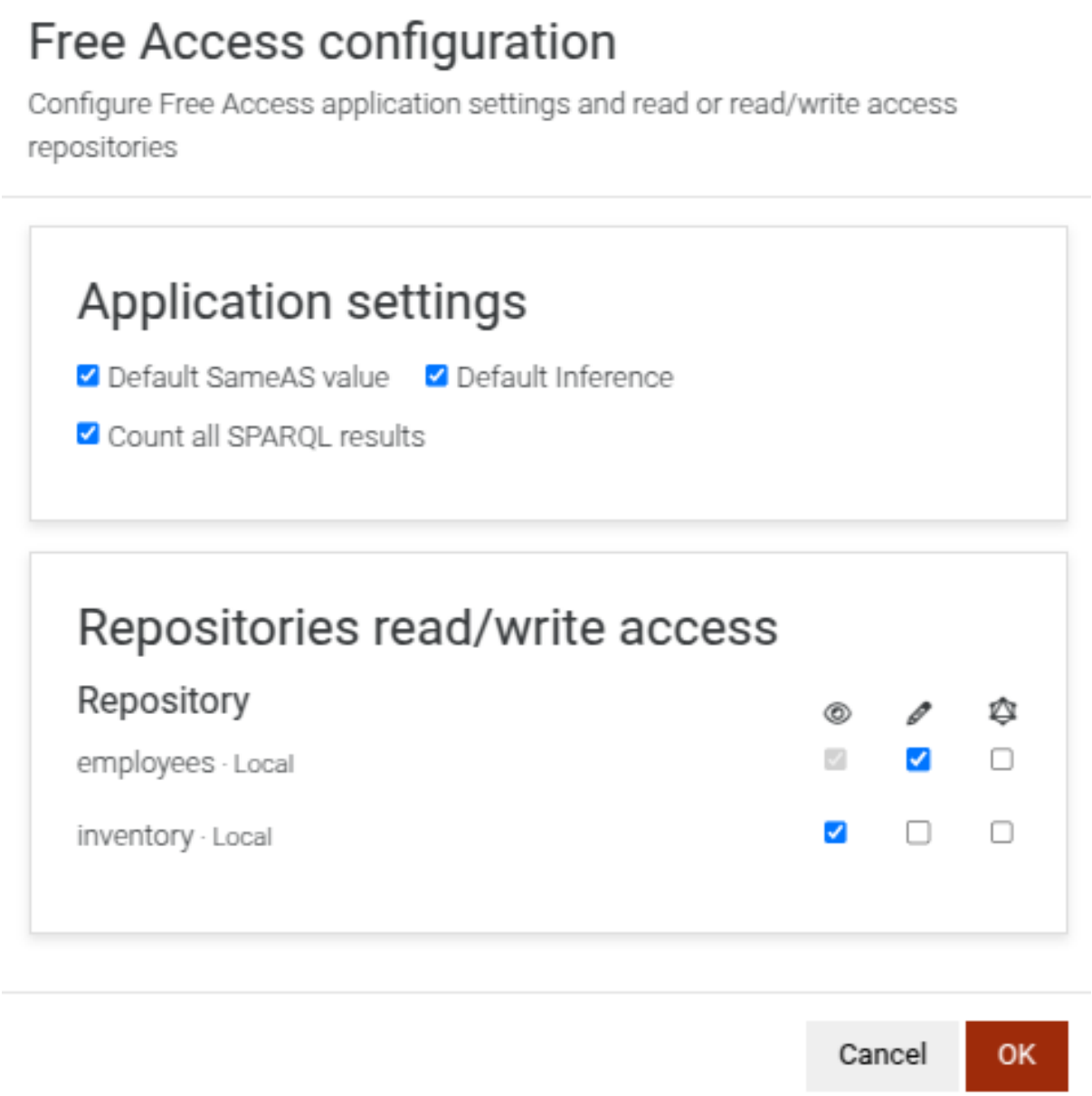
The default admin credentials are:

username: **admin**
password: **root**

Note: We recommend changing the default credentials for the admin account as soon as possible. Using the default password in production is not secure.

6.2.1.2 Free access

Once you have enabled security, you can turn on free access mode. If you click the slider associated with it, you will be shown this pop-up box:



This gives you the ability to allow unrestricted access to a number of resources without the need of any authentication. This is useful when creating a public SPARQL endpoint.

In the example above, all users will be able to read and write in the "employees" repository and only read the "inventory" repository. They will also be able to create or delete connectors and toggle plugins for the "employees" repository. If either of these two repositories had an additional check in the third column ("GraphQL only rights"), then that access would only be available via the GraphQL interface. (See this documentation's *GraphQL* section for more information on GraphDB's support for this query language.)

The Workbench settings described in *Customizing Workbench behavior* let you configure the default behavior for the GraphDB Workbench. Here, you can enable or disable the following:

- *Default sameAs value* — This is the default value for the *Expand results over owl:sameAs* option in the SPARQL editor. It is taken each time a new tab is created. Note that once you toggle the value in the editor, the changed value is saved in your browser, so the default is used only for new tabs. The setting is also reflected in the *Graph settings* panel of the Visual graph.
- *Default Inference* — Same as above, but for the *Include inferred data in results* option in the SPARQL editor. The setting is also reflected in the *Graph settings* panel of the Visual graph.

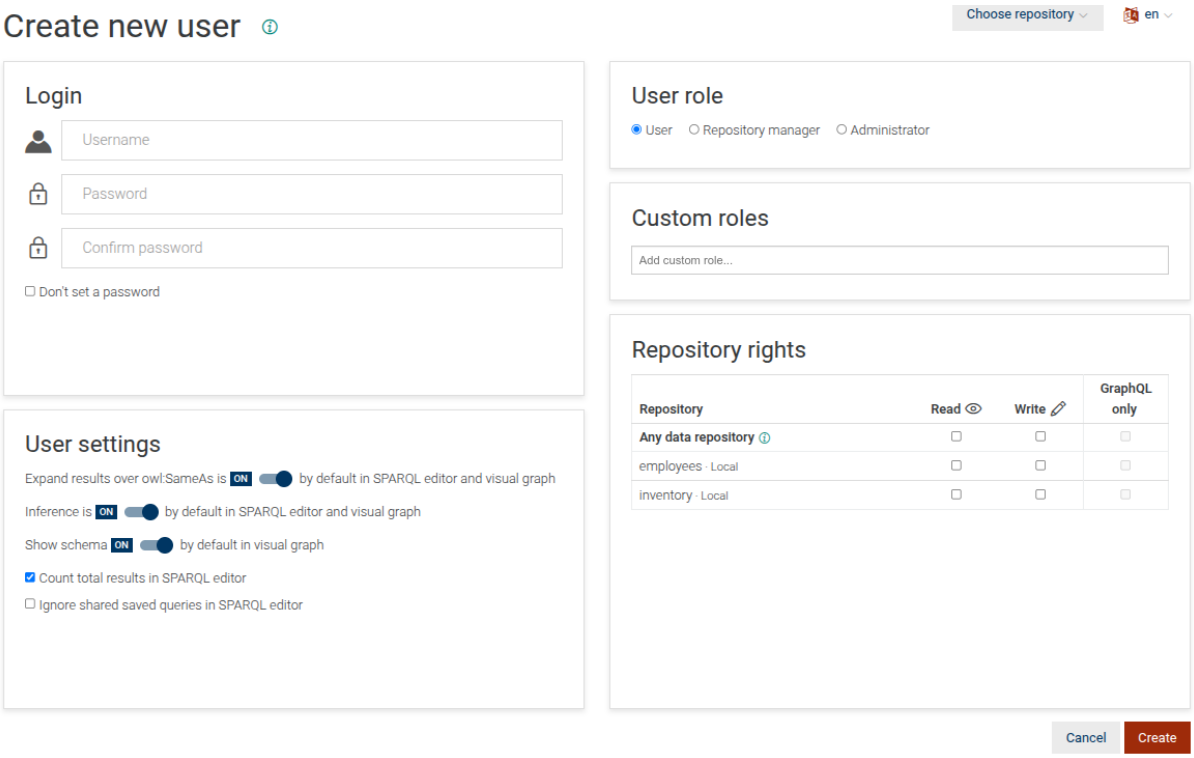
- *Count all SPARQL results* — For each query without limit sent through the SPARQL editor, an additional query is sent to determine the total number of results. This value is needed both for your information and for results pagination. In some cases, you do not want this additional query to be executed, because for example the evaluation may be too slow for your data set. Set this option to `false` in this case.

6.2.2 User management

When you are logged in as an admin user, which is the default, you can create, delete, and modify user accounts.

6.2.2.1 Create new user

To reach the *Create new user* screen, select *Setup* ▸ *Users and Access* and then click the *Create new user* button. The following shows an example of this screen:



Note: Workbench uses a similar screen for users to customize Workbench behavior. See *Customizing Workbench behavior* for more on the *User settings* section of this screen.

Each account must have a non-blank password set for it, unless *Don't set a password* has been enabled.

A user account can be assigned one of three built-in roles:

- *User* — users with this role can use GraphDB and they can save SPARQL queries, graph visualizations and user-specific server-side settings. They can also be given read and write access to specific repositories.
- *Repository manager* — in addition to the capabilities of a User role, a user with a Repository manager role also has full read and write permission to all repositories and can create, edit, and delete repositories. Repository managers can also access monitoring features and configure whether the service reports anonymous usage statistics.
- *Admin* — in addition to what Repository managers can do, a user with an Admin role can perform any server operation. This includes management of users and access, management of clusters, attaching remote locations, and viewing system information.

In the *Repository rights* section of this screen, granting an account write permission for a given repository includes read permission for that repository. If the *GraphQL only* checkbox is selected with the *Read* checkbox for a given user and repository (or for *Any data repository*), then when security is turned on for this GraphDB instance, that user will only be able read that data using the GraphQL interface. If *GraphQL only* is checked with the *Write* checkbox, the specified user will only be able to write to the indicated repository or repositories with the GraphQL interface when security is turned on. (See this documentation's *GraphQL* section for more information on GraphDB's support for this query language.)

You can grant global access to all repositories to any individual user by using the *Any data repository* checkbox.

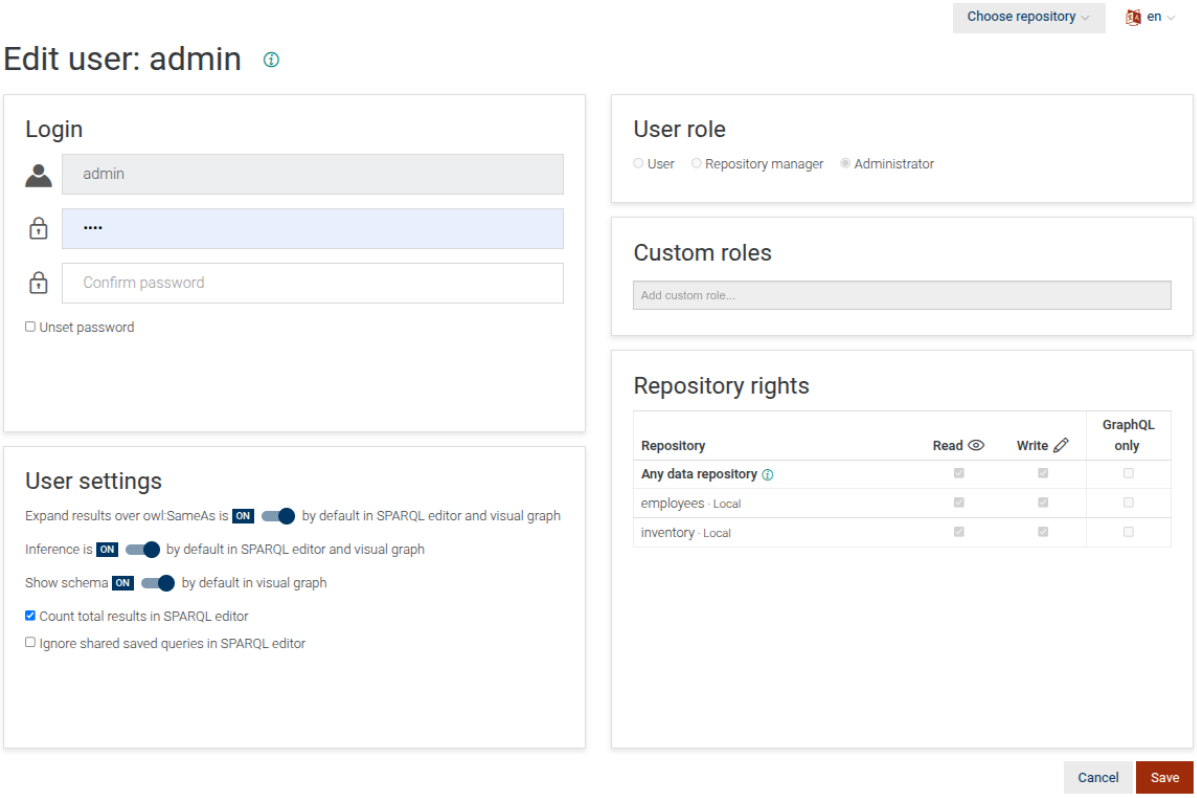
This interface also provides *Custom user roles*, which can be used together with *Fine-grained access control* for controlling access to individual RDF statements or named graphs, or *Role-Based Access Control* in GraphQL.

6.2.2.2 Delete user

To delete a user account, go to the *Setup* ▸ *Users and Access* screen and click the *Delete user* trash can icon on the row listing the account that you wish to delete. A message box will ask you to confirm before it deletes the account.

6.2.2.3 Edit user account details

On the *Setup* ▸ *Users and Access* screen, each edit icon in the *Actions* column takes you to the screen that lets you edit the configuration of the user listed on that icon's row. The following shows the edit screen for the `admin` user:



The only difference between the *Edit user* and *Create new user* screens is that in *Edit user* you cannot change the username.

The settings available through this form are the same as the settings that can be modified when *creating a new user account*, with the sole exception of the name of the account, which cannot be changed.

6.2.3 Access control

6.2.3.1 Authorization and user database

Authorization is the process of mapping a known user to a set of specific permissions. GraphDB implements Spring Security, where permissions are defined based on a combination of a URL pattern and an HTTP method. When an HTTP request is received, Spring Security intercepts it, verifies the permissions, and either grants or denies access.

User roles and permissions

GraphDB's access control is implemented using a hierarchical Role Based Access Control (RBAC) model. This corresponds to the hierarchical level of the NIST/ANSI/INCITS RBAC standard and is also known as $RBAC_1$ in older publications.

The model defines three entities:

Users Users are members of roles and acquire the permissions associated with the roles.

Roles Roles group a set of permissions and are organized in hierarchies, i.e., a role includes its directly associated permissions as well as the permissions it inherits from any parent roles.

Permissions Permissions grant access rights to execute a specific operation.

RBAC in GraphDB does not define sessions, as the security implementation is stateless. An authorized user always receives the full set of roles associated with it. Within a single API request call there is always an associated user and hence roles and permissions.

The core roles defined in GraphDB security model follow a hierarchy:

Role name	Inherits roles	Associated permissions (without the inherited ones)
ROLE_ADMIN	ROLE_REPO_MANAGER ROLE_CLUSTER	Can perform all operations, i.e., the security never rejects an operation
ROLE_REPO_MANAGER	ROLE_MONITORING	Can create, edit, and delete repositories with read and write permissions to all repositories
ROLE_MONITORING	ROLE_USER	Allows monitoring operations (queries, updates, abort query/update, resource monitoring)
ROLE_USER		Can save SPARQL queries, graph visualizations, or user-specific settings
ROLE_CLUSTER		Can perform internal cluster operations

The following repository access roles are available as well:

Role name	Associated permissions
READ_REPO_*	Read permissions to all repository IDs
WRITE_REPO_*	Write permissions to all repository IDs
READ_REPO_*xxx*	Read permissions to a given repository, where *xxx* is the repository ID
WRITE_REPO_*xxx*	Write permissions to a given repository, where *xxx* is the repository ID

Note: When providing the WRITE_REPO_xxx role for a given repository, the READ_REPO_xxx role must be provided for it as well.

Fine-grained access control can be used for controlling access below the repository level, e.g., restricting individual RDF statements or named graphs.

The GraphDB user management interface uses a simplified high level model, where each created user falls into one of three categories: a regular user, a repository manager, or an administrator. The three categories correspond directly to one of the core roles. In addition to that, regular users may be granted individual read/write rights to one or more repositories:

Inherent role and permissions	Regular user	Repository manager	Administrator
Core role	ROLE_USER	ROLE_REPO_MANAGER	ROLE_ADMIN
Bypass *FGAC* checks	no	yes	yes
Read access to a specific repository	optional	no	no
Read/write access to a specific repository	optional	no	no
Read/write access to all repositories	no	yes	yes
Create, edit, and delete repositories	no	yes	yes
Access monitoring	no	yes	yes
Manage Connectors	no	yes	yes
Manage Users and Access	no	no	yes
Manage the cluster	no	no	yes
Attach remote locations	no	no	yes
View system information	no	no	yes

Custom roles

GraphDB has a facility for defining *custom roles* that can be used together with *Fine-grained access control* for controlling access to individual RDF statements or named graphs.

Built-in users and roles

GraphDB has two special internal users that are required for the functioning of the database. These users cannot be seen or modified via user management.

Username	Associated roles	Description
<cluster user>	ROLE_CLUSTER READ_REPO_* WRITE_REPO_*	Used for cluster-internal communication between cluster nodes.
<free access user>	None by default but administrators may grant access to one or more repositories.	The user associated with anonymous access if anonymous access is enabled. Check the security documentation to see how to configure *Free access*.

GraphDB supports three types of user databases used for authentication and authorization, explained in detail below: Local, LDAP, and OAuth. Each of them contains the information about who the user is, where they come

from, and what type of rights and roles they have. The database may also store and validate the user's credentials, if that is required.

Only one database is active at a time. When one is selected, all available users are provided from that database.

The default database is Local.

Local user database

As mentioned above, this is the default security access provider. The local database stores usernames, encrypted passwords, assigned roles and user settings. Passwords are encrypted using the bcrypt algorithm.

The local database is located in the `settings.js` file under the GraphDB `data` directory. If you are worried about the security of this file, we recommend encrypting it (see *Encryption at rest*).

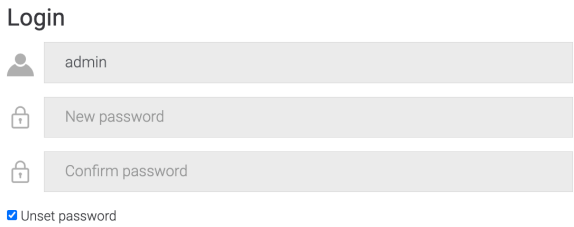
The local user database does not need to be configured but can be explicitly specified with the following property:

```
graphdb.auth.database = local
```

Default admin user

A fresh installation of GraphDB comes with a single default user whose username is `admin` and default password is `root`. This user cannot be deleted or demoted to any of the non-administrator levels. It is recommended to change the default password at earliest convenience in order to avoid undesired access by a third party.

If you wish to disable the default admin user, you can unset its password from *Setup ▸ My Settings* in the GraphDB Workbench.



Warning: If you unset the password for any user and then enable security, that user will not be able to log into GraphDB. The only way to log in would be through *OpenID* or *Kerberos* authentication.

LDAP user database

Note: Using LDAP requires a *GraphDB Enterprise or Standard license*.

Tip: See also the configuration examples for *Basic/GDB + LDAP*, *OpenID + LDAP*, and *Kerberos + LDAP*.

Lightweight Directory Access Protocol (LDAP) is a lightweight client-server protocol for accessing directory services implementing X.500 standards. All its records are organized around the LDAP Data Interchange Format (LDIF), which is represented in a standard plain text file format.

When LDAP is enabled and configured, it replaces the local database and GraphDB security will use the LDAP server to provide authentication and authorization. An internal user settings database is still used for storing user settings. This means that you can use the Workbench or the GraphDB API to change them. All other administration operations need to be performed on the LDAP server side.

Note: As of GraphDB version 9.5 and newer, local users will no longer be accessible when using LDAP.

LDAP needs to be configured in the `graphdb.properties` file.

Enable LDAP with the following property:

```
graphdb.auth.database = ldap
```

When LDAP is turned on, the following security settings can be used to configure it.

Property	Description	Example value
`graphdb.auth.ldap.url` (**required**)	LDAP endpoint	`ldap://<my-openldap-server>:389/<partition>`
`graphdb.auth.ldap.user.search.base`	Query to identify the directory where all authenticated users are located.	<empty>
`graphdb.auth.ldap.user.search.filter` (**required**)	Matches the attribute to a GraphDB username.	`(cn={0})`
`graphdb.auth.ldap.user.search.subtree`	Enable subtree search for users. Default is `true`.	
`graphdb.auth.ldap.role.search.base` (**required**)	Query to identify the directory where roles/groups for authenticated users are located.	<empty>

`graphdb.auth.ldap.role.search.filter` (**required**)	Authorize a user by matching the manner in which they are listed within the group. The property value supports two placeholders, `{0}` and `{1}`, each with a different meaning:
	• The placeholder `{0}` is replaced with the full LDAP distinguished name, e.g., `cn=johnsmith,o=example,o=com`.
	• The placeholder `{1}` is replaced with just the common name (the `cn` field), e.g., "johnsmith". Typically, users are mapped to groups using only the common name, so we recommend setting the value to `{1}`.
	Tip: It may be useful to enable debug logging for LDAP by adding the following at the end of the `conf/logback.xml` file in the GraphDB distribution:
	`<logger name="com.ontotext.forest.security.provider.ldap" level="DEBUG"/>`
	`<logger name="org.springframework.security.ldap" level="DEBUG"/>`
	This will print additional LDAP messages in the log showing what is queried and what is returned.
		`(uniqueMember={1})`

`graphdb.auth.ldap.role.search.attribute`	The attribute to identify the common name.	`cn` (default)
`graphdb.auth.ldap.role.map.administrator`	Map a single LDAP group to GDB administrator role.	`my-group-name`

Providing access to a family of repositories that share a naming convention

With `graphdb.auth.ldap.role.map.repository.read.<my-repo>` and `graphdb.auth.ldap.role.map.repository.write.<my-repo>`, you can also use `*` wildcards in combination with a prefix to provide read or write access to a family of repositories that share a naming convention — a common prefix in their name, followed by a differentiator.

The `*` wildcard matches zero or more characters and can only be used with a prefix — `*` wildcards in the beginning or middle of a repository name do not work and will throw exceptions. All other characters are treated literally.

For example, `graphdb.auth.ldap.role.map.repository.read.project-*` will grant access read to all repositories whose name begins with the `project-` prefix, such as `project-repository1`, `project-repository2`, `project-repository2-alternative` and even `project-`, but it would not provide access to a repository named `repository3-project`.

Using prefixes and suffixed with LDAP group names

The `graphdb.auth.ldap.group.prefix` and `graphdb.auth.ldap.group.suffix` properties filter LDAP groups before mapping them to GraphDB repository roles. Only groups matching the configured prefix and suffix pattern are processed. Groups that match that pattern have the matching prefix and suffix stripped before role assignment.

If both `graphdb.auth.ldap.group.prefix` and `graphdb.auth.ldap.group.suffix` are configured, only groups that match both will be processed. In other words, if you have configured both a prefix and a suffix, and have an LDAP group that matches only one of them instead of both (by having only a prefix or a suffix), it will not be processed.

Note: This filtering does not apply to Administrator and Repository manager role mappings (`graphdb.auth.ldap.role.map.administrator` and `graphdb.auth.ldap.role.map.repositoryManager`), which continue to work with the full group name without prefix or suffix stripping.

Mapping user type roles

GraphDB has three standard user roles: Administrator, Repository manager, and User. Every user authenticated over LDAP will be assigned one of these roles.

Mapping the Administrator role

Set the following property to the LDAP group that must receive this role:

```
graphdb.auth.ldap.role.map.administrator = gdbadmin
```

Mapping the Repository manager role

Set the following property to the LDAP group that must receive this role:

```
graphdb.auth.ldap.role.map.repositoryManager = gdbrepomanager
```

Mapping the User role

Unless a user has been assigned the Administrator or Repository manager role, they will receive the User role automatically.

OAuth user database

Note: Using OAuth requires a *GraphDB Enterprise or Standard license*.

Tip: See also the configuration example for *OpenID + OAuth*.

OAuth is an open-standard authorization protocol for providing secure delegated access as a way for users to grant websites/applications access to their information on other websites/applications without sharing their initial login credential. OAuth is centralized, which means only the authorization server owns user credentials.

Note: OAuth requires OpenID for authentication, and the authorization comes from an OAuth claim. Direct password authentication with GraphDB (e.g., basic or using the Workbench login form) is not possible.

When OAuth is enabled and configured, it replaces the local database and GraphDB security will use only the OAuth claims to provide authorization. An internal user settings database is still used for storing user settings. This means that you can use the Workbench or the GraphDB API to change them. All other administration operations need to be performed in the OpenID/OAuth provider.

Enable OAuth authorization with the following property:

```
graphdb.auth.database = oauth
```

When OAuth authorization is enabled, the following property settings can be used to configure it:

Property	Description	Example value
`graphdb.auth.oauth.roles_claim` **(required)**	OAuth roles claim. The field from the JWT token that will provide the GraphDB roles, as a JSONPath expression. No default value. See *JSONPath for roles claims examples* for specific examples covering several use-cases.	`roles` (field name without dots) `roles.graphdb` (nested field inside another field) `$['roles.graphdb']` (field name with dots)
`graphdb.auth.oauth.default_roles` **(required)**	OAuth default roles to assign. It may be convenient to always assign certain roles without listing them in the roles claim. The value is a comma-delimited list of GraphDB roles. The default value is the empty list.	`ROLE_USER`
`graphdb.auth.oauth.roles_prefix`	OAuth roles prefix to strip. The roles claim may provide the GraphDB roles with some prefix, for example — `GDB_ROLE_USER`. The prefix will be stripped when the roles are mapped. The default value is the empty string.	`GDB_`
`graphdb.auth.oauth.roles_suffix`	OAuth roles suffix to strip. The roles claim may provide the GraphDB roles with some suffix, for example — `ROLE_USER_GDB`. The suffix will be stripped when the roles are mapped. The default value is the empty string.	`_GDB`
`graphdb.auth.oauth.bare_custom_roles`	When true, *Custom user roles* from OAuth tokens are mapped without requiring the `CUSTOM_` prefix. The default value is `false`.	`false`
`graphdb.auth.openid.require_audience`	When true, JSON web tokens (JWTs) must include a valid 'aud' claim matching the configured audience. When false, tokens without 'aud' are accepted, optionally validated against the client ID. The default value is `true`	`true`

JSONPath for roles claims examples

Tip: Do not be concerned with learning the full JSONPath specification as most JSONPath expressions you will need to configure the roles claim are intuitive.

Let's look at three common use-cases:

`field_name_without_dots` Use this expression when the claims are directly contained in the top-level claims under a key name **without** dots. For example, use `roles` if the token contains a "roles" key, as illustrated below:

```
{
    "roles": [
        "ROLE_USER", "WRITE_REPO_*", "READ_REPO_*"
    ]
}
```

`nested.key.with.dots` Use this expression when the claims are contained inside a non-top-level object where

each part separated by dots signifies a different level. For example, use `roles.graphdb` if the token contains a "roles" key with a nested "graphdb" key, as illustrated below:

```
{
    "roles": {
        "graphdb": [
            "ROLE_USER", "WRITE_REPO_*", "READ_REPO_*"
        ]
    }
}
```

`$['field.name.with.dots']` Use this expression when the claims are directly contained in the top-level claims under a key name **with** dots. For example, use `$['roles.graphdb']` if the token contains a "roles.graphdb" key, as illustrated below:

```
{
    "roles.graphdb": [
        "ROLE_USER", "WRITE_REPO_*", "READ_REPO_*"
    ]
}
```

Tip: Use the JSONPath Online Evaluator to test your JSONPath expression with your decoded token.

Case-insensitive user database

GraphDB can use case-insensitive validation for user accounts so that users can log in regardless of the case used at login time. For example, if the user database contains a user "john.smith", they can log in using any of these:

- john.smith
- John.Smith
- JOHN.SMITH
- JOHN.smitH

This is controlled via the boolean config property `graphdb.auth.database.case_insensitive`. It is optional and `false` by default.

When using the local database, it is enough to just set `graphdb.auth.database.case_insensitive = true`. When using LDAP, you can use the case-insensitive option to control how group names are matched when mapping LDAP groups to GraphDB roles.

When using an external user database (LDAP, OpenID), the external database must support case-insensitive login as well.

6.2.3.2 Authentication methods

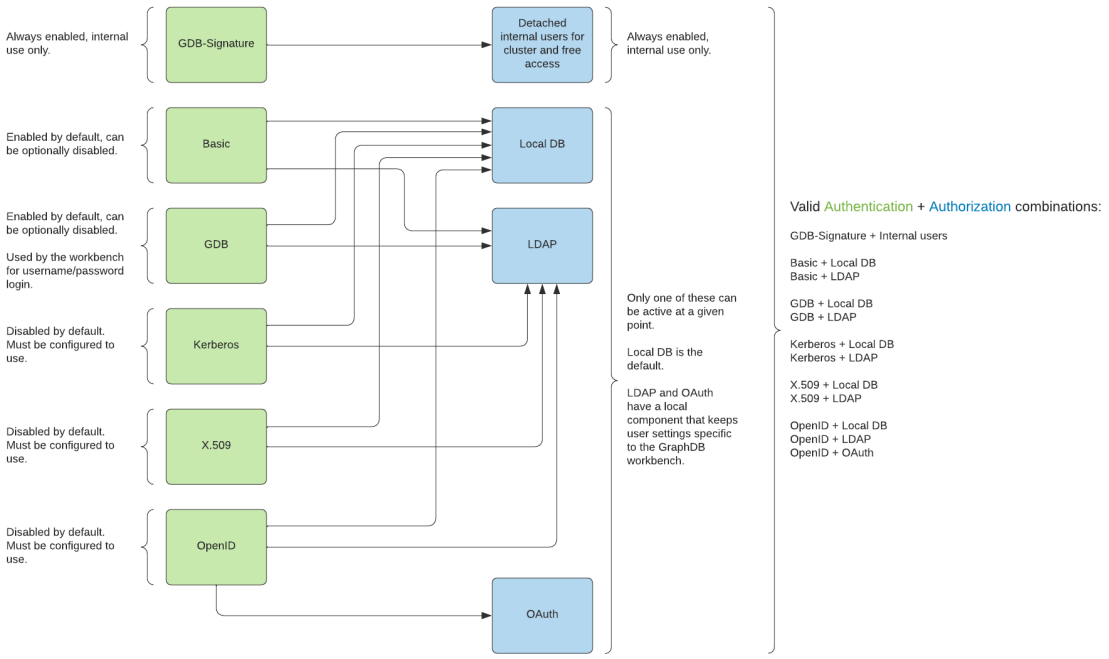
Whenever a client connects to GraphDB, a security context is created. Each security context is always associated with a single authenticated user or a default anonymous user when no credentials have been provided.

Authentication is the process of mapping this security context to a specific user. Once the security context is mapped to a user, a set of permissions can be associated with it, using authorization.

When GraphDB security is ON, the following authentication methods are available:

- Basic authentication: The username and password are sent in a header as plain text (usually used when using the RDF4J client, or via Java when run with cURL). Enabled by default (can be optionally disabled).
- GDB: Token-based authentication used by the Workbench for username/password login. This login method is also available through the REST API. Enabled by default (can be optionally disabled).

- Kerberos: Highly secure single sign-on protocol that uses tickets for authentication. Disabled by default (must be configured to be enabled).
- X.509 certificate authentication: When a certificate is signed by a trusted authority, or is otherwise validated, the device holding the certificate can validate documents. Disabled by default (must be configured to be enabled).
- OpenID: Single sign-on method that allows accessing GraphDB without the need for creating a new password. Its biggest advantage is the delegation of the security outside the database. Disabled by default (must be configured to be enabled).



All five authentication providers — Basic, GDB, OpenID, X.509, and Kerberos — can be combined with both a local and an LDAP database. The only provider that can be combined with OAuth is OpenID, as OAuth is an extension of the latter.

There is also an additional authentication provider, the GDB Signature. It is for internal use only, works with a detached internal cluster user, and is always enabled. This is the built-in cluster security that uses tokens similar to those used for logging in from the Workbench.

The following combinations of authentication provider and user database are possible:

Authentication provider	User database
Basic authentication	Local DB LDAP
Kerberos	Local DB LDAP
X.509 certificate	Local DB LDAP
GDB	Local DB LDAP
OpenID	Local DB LDAP OAuth

We will look at each of the above in greater detail in the following sections.

Basic authentication

Basic authentication is a method for an HTTP client to provide a username and password when making a request. The request contains a header in the form of `Authorization: Basic <credentials>`, where `<credentials>` is the Base64 encoding of the username and password joined by a single colon, e.g.:

```
Authorization: Basic YWRtaW46cm9vdA==
```

> **Warning:** Basic authentication is the least secure authentication method. Anyone who intercepts your requests will be able to reuse your credentials indefinitely until you change them. Since the credentials are merely base-64 encoded, they will also get your username and password. This is why it is very important to always use *encryption in transit*.

GDB authentication

GDB authentication is a method for an HTTP client to obtain a token in advance by supplying a username and password, and then send the token with every HTTP request that requires authentication. The token must be sent as an HTTP header in the form of `Authorization: GDB <token>`, where `<token>` is the actual token.

This authentication method is used by the GraphDB Workbench when a user logs in by typing their username and password in the Workbench.

Note: Anyone who intercepts a GDB token can reuse it until it expires. To prevent this, we recommend to always enable *encryption in transit*.

It is also possible to obtain a token via the REST API and use the token in your own HTTP client to authenticate with GraphDB, e.g. with cURL:

1. Log in and obtain a token:

```
curl -X POST -I 'http://localhost:7200/rest/login' -H 'Content-type: application/json' -d '{
     "username": "admin",
     "password": "root"
}'
```

   The token will be returned in the Authorization header. It can be copied as is and used to authenticate other requests.

2. Use the returned token to authenticate with GraphDB:

```
curl -H 'Authorization: GDB eyJ1c2VybmFtZSI6ImFkbWlu...' http://localhost:7200/repositories/myrepo/
↪size
```

GDB tokens are signed with a private key and the signature is valid for a limited period of time. If the private key changes or the signature expires, the token is no longer valid and the user must obtain a new token. The default validity period is 30 days. It can be configured via the `graphdb.auth.token.validity` property that takes a single number, optionally suffixed by the letters *d* (days), *h* (hours) or *m* (minutes) to specify the unit. If no letter is provided, then days are assumed. For example, `graphdb.auth.token.validity = 2d` and `graphdb.auth.token.validity = 2` will both set the validity to two days.

Note: During the token validity period, if the password is changed the user will still have access to the server. However, if the user is removed, the token will stop working.

The private key used to sign the GDB tokens is generated randomly when GraphDB starts. This means that after a restart, all tokens issued previously will expire immediately and users will be forced to login again. To avoid that, you can set a secret to derive a static private key by setting the following property:

```
graphdb.auth.token.secret = <my-secret>
```

Treat the secret as any password, it must be sufficiently long and not easily guessable.

Note: The token secret is used to sign the internal cluster communication and needs to be the same on all cluster nodes.

OpenID authentication

Note: Using OpenID requires a *GraphDB Enterprise or Standard license*.

Tip: See also the configuration examples for *OpenID + local users*, *OpenID + LDAP*, and *OpenID + OAuth*.

Single sign-on over the OpenID protocol enables you to log in just once and access all internal services. From a security standpoint, it provides a more secure environment, because it minimizes the number of places where a password is processed.

When OpenID is used for authentication, the authorization may come from the local user database, LDAP, or OAuth. Direct password authentication with GraphDB is possible only with the local database or LDAP, and can be optionally disabled.

OpenID needs to be configured from the `graphdb.properties` file. Enable it with the following property:

```
graphdb.auth.methods = basic, gdb, openid
```

The default value is `basic, gdb`.

Provide only `openid` if password-based login methods (Basic and GDB) are not needed, or if you combine OpenID with the OAuth user database.

When OpenID authentication is enabled, the following property settings can be used to configure it:

Property	Description	Example value
`graphdb.auth.openid.issuer` (**required**)	OpenID issuer URL used to derive keys, endpoints, and token validation. No default value.	`https://accounts.example.com`
`graphdb.auth.openid.client_id` (**required**)	OpenID client ID used to authenticate and validate tokens. No default value.	`<my-client-id>`
`graphdb.auth.openid.username_claim` (**required**)	OpenID claim to use as the GraphDB username. No default value.	`email`
`graphdb.auth.openid.auth_flow` (**required**)	OpenID authentication flow: code, code_no_pkce, implicit. The recommended value is `code` if the OpenID provider supports it with PKCE without a client secret. No default value.	`code`
`graphdb.auth.openid.token_type` (**required**)	OpenID token type to send to GraphDB. The available values are `access` and `id`. Use the access token if it is a JWT token, otherwise use the id token. No default value.	`access`
`graphdb.auth.openid.token_issuer`	OpenID expected issuer URL in tokens, used to validate tokens. The default is the same as the actual issuer URL.	`https://accounts.example.com/custom`
`graphdb.auth.openid.token_audience`	OpenID expected audience in tokens, used to validate tokens. The default is the same as the client ID.	`<my-audience>`
`graphdb.auth.openid.header_types`	OpenID authentication type to specify the allowed `"typ"` values in JWT headers. The value is a comma-delimited list, where `null` is a special value, interpreted as no `"typ"` field in the header. The default value is `jwt, at+jwt, null`.	`jwt`
`graphdb.auth.openid.authorize_parameters`	OpenID extra parameters for the authorize endpoint. Some OpenID providers require additional parameters sent to the authorize endpoint (e.g., `resource=xxx`). This is a URL-encoded string where each parameter-value pair is delimited by `&`. The string will be appended to the rest of the authorize URL parameters. The default value is the empty string.	`param1=value%201&param2=value%202`
`graphdb.auth.openid.proxy`	OpenID uses GraphDB as proxy for the JWKS URL and token endpoints. This can be used to bypass an OpenID provider without a proper CORS configuration. The value is a boolean `true/false`. `False` by default.	`false`
`graphdb.auth.openid.extra_scopes`	OpenID extra scopes to request. Multiple scopes can be specified by separating them with a space. By default, GraphDB requests only the 'openid' scope and, if sup-	`profile email`

Note: Logging out in this mode when using the GraphDB Workbench only deletes the GraphDB session without logging you out from your provider account.

Configuring the OpenID provider

The OpenID provider needs to be configured as well, as the GraphDB Workbench will use its own root browser URL, e.g., `https://graphdb.example.com:7200/` (note the terminating slash) as the `redirect_uri` parameter when it redirects the browser to the authorization endpoint. Once the login is completed at the remote end, OIDC mandates that the identity provider redirects back to the supplied `redirect_uri`.

Typically, the allowed values for `redirect_uri` must be registered with the OpenID provider.

Kerberos authentication

Note: Using Kerberos requires a *GraphDB Enterprise or Standard license*.

Tip: See also the example configurations for *Kerberos + local users* and *Kerberos + LDAP*.

Kerberos is a highly secure single sign-on protocol that uses tickets for authentication, and avoids storing passwords locally or sending them over the Internet. The authentication mechanism involves communication between a trusted third-party connection encrypted with symmetric-key cryptography. Although considered a legacy technology, Kerberos is still the default single sign-on mechanism in big Windows-based enterprises, and is an alternative of OpenID authentication.

The basic support for authentication via Kerberos in GraphDB involves:

- Validation of SPNEGO HTTP Authorization tokens. For example:

```
Authorization: Negotiate XXXXXXX
```

- Extraction of the username from the SPNEGO token and matching the username against a user from the local database or a user from LDAP.

SPNEGO is the mechanism that integrates Kerberos with HTTP authentication.

After the token is validated and matched to an existing user, the process continues with authorization (assigning user roles) via the existing mechanism.

Using Kerberos this way is equivalent to authenticating via Basic, GDB, or OpenID.

Configuring Kerberos in GraphDB

In order to validate incoming SPNEGO tokens, the Spring Security Kerberos module needs a Kerberos *keytab* (a set of keys associated with a particular Kerberos account) and a *service principal* (the username of the associated Kerberos account). This account is used only to validate and decrypt the incoming SPNEGO tokens and is not associated with any user in GraphDB. See more on how to create a `keytab` file in the Kerberos documentation.

Enable Kerberos with the following property:

```
graphdb.auth.methods = basic, gdb, kerberos
```

The default value is `basic, gdb`.

Kerberos is configured via several properties:

Property	Description	Example value
`graphdb.auth.kerberos.keytab` (**required**)	Full or relative (to the GraphDB config directory) path to where the keys of the Kerberos service principal are stored. Required if Kerberos is enabled.	`graphdb-http.keytab`
`graphdb.auth.kerberos.principal` (**required**)	Name of the Kerberos service principal. Required if Kerberos is enabled.	`HTTP/data.example.com@EXAMPLE.COM`
`graphdb.auth.kerberos.debug`	Whether some of the Spring Kerberos classes print extra messages related to Kerberos.	`false`

In addition, you might want to specify a custom `krb5.conf` file via the `java.security.krb5.conf` property but Java should be able to pick up the default system file automatically.

User matching

Kerberos principals (usernames) need to be matched to GraphDB usernames. A Kerberos principal consists of a username, followed by @, followed by a realm. The realm looks like a domain name and is usually written out in capital letters. The principals are converted by simply dropping the @ sign and the realm. However, the realm from incoming SPNEGO tokens must match the realm of the service principal. Some examples:

Service principal	Principal from SPNEGO token	Username in GraphDB
`HTTP/data.example.com@EXAMPLE.COM`	`john@EXAMPLE.COM`	john
`HTTP/data.example.com@EXAMPLE.COM`	`john@FOO.EXAMPLE.COM`	Invalid authentication because of realm mismatch

Using SPNEGO tokens with GraphDB

There are various ways to use SPNEGO when talking to GraphDB as a client. All methods add the Kerberos/SPNEGO authentication in the HTTP client used by the RDF4J libraries.

Native method

The native method does not require any third-party libraries and relies on the built-in Kerberos capabilities of Java and Apache's HttpClient. However, it is a bit cumbersome to use since it requires wrapping calls into an authentication context. This method supports only non-preemptive authentication, i.e., the GraphDB server must explicitly say it needs Kerberos/SPNEGO by sending a `WWW-Authenticate: Negotiate` header to the client.

Third-party library method

There is a third-party library called kerb4j, which makes some things easier. It does not require wrapping the execution into an authentication context and supports preemptive authentication, i.e., sending the necessary headers without asking the server if it needs authentication.

Both methods are illustrated in this example project.

X.509 certificate authentication

X.509 is a digital certificate based on the widely accepted International Telecommunications Union (ITU) X.509 standard, which defines the format of public key infrastructure (PKI) certificates. Some of its advantages include:

- Increased security compared to traditional username and password combinations.
- Streamlined authentication as certificates eliminate the need to remember username and password combinations.
- Ease of deployment as certificates are stored locally and are implemented without needing any extra hardware.

This authentication method can be used with the local users and the LDAP authorization databases. Direct password authentication with GraphDB is possible only with local users or LDAP, and can be optionally disabled.

1. Enable X.509 certificate authentication with the following `graphdb.properties` file property. The default value is `basic, gdb`. Provide only `x509` if password-based login methods (basic and gdb) will not be used.

```
graphdb.auth.methods = basic, gdb, x509
```

2. Enable local or LDAP authorization. The default value of the property is `local`, corresponding to the local user database. If LDAP is the chosen authorization database, enable it via the property below and then *configure it*.

```
graphdb.auth.database = ldap
```

3. Provide a regular expression to extract the username from the certificate. The default is `CN=(.*?)(?:,|$)`. If you want to provide a custom expression, uncomment the following and edit it.

```
graphdb.auth.methods.x509.subject.dn.pattern = CN=(.*?)(?:,|$)
```

4. To implement server-side X.509 authentication, *enable SSL*.
5. (optional) To implement the client-side authentication, we need a truststore that holds the certificate authorities (CA) for the intended client certificates. To set up a truststore different from the default JRE one, uncomment the following properties and set `truststoreFile` and `truststorePassword` to their actual values.

```
graphdb.connector.ssl.truststoreFile = <path-to-custom-truststore-file>
graphdb.connector.ssl.truststorePassword = <secret>
```

6. Next, we will configure the X.509 certificate revocation status check.
   a. One of two checks can be performed: Online Certificate Status Protocol (OCSP) and Certificate Revocation List Distribution Point (CRLDP) check. Both are `true` by default when commented out. This means that if you want to enable one of them, you need to uncomment the other one and set it to `false`.

```
graphdb.auth.methods.x509.ocsp = true
graphdb.auth.methods.x509.crldp = true
```

   b. There is also a third option — setting a Certificate Revocation List (CRL) to Tomcat, which will allow revocation checks for certificates that do not provide an Authority Information

Access (AIA) extension or can serve as an alternative in the event of OCSP or CRLDP responders downtime. Uncomment and set the property:

```
graphdb.auth.methods.x509.crlFile = <path-to-certificate-revocation-list>
```

Note: If all three methods are provided, the order of precedence in which GraphDB will look for them is:

1. Online Certificate Status Protocol (OCSP) check
2. Certificate Revocation List Distribution Point (CRLDP) check
3. Certificate Revocation List (CRL) check

Using X.509 certificate authentication with cURL

The method can also be configured via cURL request.

To send a client certificate to the server when communicating over HTTPS or FTPS protocol, you can use the `-E` or `--cert` command-line switch. The client certificate must be in PKCS#12 format for Secure Transport or `.pem` format if using any other mechanism.

```
curl -E cerfile.crt 'https://<base_url>/<graphdb_endpoint>'
```

```
curl --cert cerfile.crt 'https://<base_url>/<graphdb_endpoint>'
```

To bypass certificate validation, pass the `-k` or `--insecure` flag to cURL. This will tell cURL to ignore certificate errors and accept insecure certificates without complaining about them.

```
curl -k --cert cerfile.pem --key cerfile.key 'https://<base_url>/<graphdb_endpoint>'
```

Using X.509 certificate authentication with HTTP headers

GraphDB can be configured to extract and resolve x509 client certificates from the HTTP request headers. This is particularly useful when GraphDB is deployed behind a load balancer that does TLS termination. In such cases, the load balancer could be configured to do TLS passthrough which sends the client certificates inside an HTTP header allowing GraphDB to complete the mutual authentication flow.

X.509 authentication with HTTP headers is configured with the following properties:

Property	Description	Default value
`graphdb.auth.methods.x509.headers`	Enables X.509 mutual authentication to resolve client certificates from the HTTP request headers.	`false`
`graphdb.auth.methods.x509.headers.clientCert`	Name of the HTTP header used for sending and extracting client certificates.	`ssl_client_cert`
`graphdb.auth.methods.x509.headers.escapedClientCert`	Name of the HTTP header used for sending and extracting client certificates when the certificates are being encoded by the load balancer. Takes precedence over `graphdb.auth.methods.x509.headers.clientCert`.	`ssl_client_escaped_cert`

The minimal setup to enable X509 authentication with HTTP request headers is to configure:

```
graphdb.auth.methods.x509.headers=true
```

Usually, using the configuration above should be sufficient. However, if a load balancer or a proxy uses a different request header to send client certificates, the `graphdb.auth.methods.x509.headers.clientCert` property must also be configured to match the header name. In case the certificates are sent encoded, the `graphdb.auth.methods.x509.headers.escapedClientCert` property must be configured instead.

For example, if the load balancer sends certificates in a header named `X-Custom-TLS-Header`, you need to configure:

```
graphdb.auth.methods.x509.headers=true
graphdb.auth.methods.x509.headers.clientCert=X-Custom-TLS-Header
```

6.2.3.3 Example configurations

This is a list of example configurations for some of the possible combinations of authentication methods (Basic, GDB, OpenID, X.509 certificate, and Kerberos) with the three supported user databases for authorization (Local, LDAP, and OAuth).

Tip: The OpenID, Kerberos and LDAP part of the examples is identical in all cases but is repeated for convenience.

Basic/GDB + LDAP

Example configuration of Basic and GDB authentication + LDAP authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ BASIC AUTHENTICATION AND GDB AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~

# The methods basic and gdb are active by default but may be provided explicitly as such:
graphdb.auth.methods = basic, gdb

# ~~~~~~~~~~~~~~~~~~~
```

(continued from previous page)

```
graphdb.auth.ldap.bind.userDn = uid=userId,ou=people,dc=example,dc=org
graphdb.auth.ldap.bind.userDn.password = 123456
```

OpenID + local users

Example configuration of OpenID authentication + local user database authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ OPENID AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Enable OpenID authentication.
graphdb.auth.methods = openid
#  or alternatively with enabled Basic and GDB password authentication:
#graphdb.auth.methods = basic, gdb, openid

# OpenID issuer URL, used to derive keys endpoints and token validation.
graphdb.auth.openid.issuer = https://accounts.example.com

# OpenID client ID used to authenticate and validate tokens.
graphdb.auth.openid.client_id = my-client-id

# OpenID claim to use as the GraphDB username.
graphdb.auth.openid.username_claim = email

# OpenID authentication flow: code, code_no_pkce or implicit.
graphdb.auth.openid.auth_flow = code

# OpenID token type to send to GraphDB.
graphdb.auth.openid.token_type = access

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LOCAL USER AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# The local database is the default setting but it may be set explicitly as such:
graphdb.auth.database = local
```

OpenID + LDAP

Example configuration for OpenID authentication + LDAP authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ OPENID AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Enable OpenID authentication.
graphdb.auth.methods = openid
#  or alternatively with enabled Basic and GDB password authentication:
#graphdb.auth.methods = basic, gdb, openid

# OpenID issuer URL, used to derive keys endpoints and token validation.
graphdb.auth.openid.issuer = https://accounts.example.com

# OpenID client ID used to authenticate and validate tokens.
graphdb.auth.openid.client_id = my-client-id

# OpenID claim to use as the GraphDB username.
graphdb.auth.openid.username_claim = email

# OpenID authentication flow: code, code_no_pkce or implicit.
graphdb.auth.openid.auth_flow = code
```

(continues on next page)

(continued from previous page)

```
# OpenID token type to send to GraphDB.
graphdb.auth.openid.token_type = access

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LDAP AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on ldap authentication and configure the server.
graphdb.auth.module = ldap
graphdb.auth.ldap.url = ldap://localhost:10389/dc=example,dc=org

# Permit access for all users that are part of the "people" unit of the fictional "example.org"
↪organization.
graphdb.auth.ldap.user.search.base = ou=people
graphdb.auth.ldap.user.search.filter = (cn={0})

# Make all users in the Administration group GraphDB administrators as well.
graphdb.auth.ldap.role.search.base = ou=groups
graphdb.auth.ldap.role.search.filter = (member={1})
graphdb.auth.ldap.role.map.administrator = Administration

# Make all users in the Management group GraphDB Repository Managers as well.
graphdb.auth.ldap.role.map.repositoryManager = Management

# Enable all users in the Readers group to read the my_repo repository.
graphdb.auth.ldap.role.map.repository.read.my_repo = Readers

# Enable all users in the Writers group to write and read the my_repo repository.
graphdb.auth.ldap.role.map.repository.write.my_repo = Writers

# Required for accessing a LDAP server that does not allow anonymous binds and anonymous access.
graphdb.auth.ldap.bind.userDn = uid=userId,ou=people,dc=example,dc=org
graphdb.auth.ldap.bind.userDn.password = 123456
```

OpenID + OAuth

Example configuration for OpenID authentication + OAuth authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ OPENID AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Enable OpenID authentication.
graphdb.auth.methods = openid

# OpenID issuer URL, used to derive keys endpoints and token validation.
graphdb.auth.openid.issuer = https://accounts.example.com

# OpenID client ID used to authenticate and validate tokens.
graphdb.auth.openid.client_id = my-client-id

# OpenID claim to use as the GraphDB username.
graphdb.auth.openid.username_claim = email

# OpenID authentication flow: code, code_no_pkce or implicit.
graphdb.auth.openid.auth_flow = code

# OpenID token type to send to GraphDB.
graphdb.auth.openid.token_type = access

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ OAUTH AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# OAuth roles claim. The field from the JWT token that will provide the GraphDB roles, as a JSONPath
↪expression.
```

(continues on next page)

(continued from previous page)

```
# Nested roles can be accessed with a JSONPath expression using dots between the nested fields, e.g.,
↪"roles.graphdb"
graphdb.auth.oauth.roles_claim = roles

# OAuth roles prefix to strip. The roles claim may provide the GraphDB roles with some prefix, e.g., GDB_
↪ROLE_USER.
# The prefix will be stripped when the roles are mapped.
graphdb.auth.oauth.roles_prefix = GDB_

# OAuth default roles to assign. It may be convenient to always assign certain roles without listing them
↪in the roles claim.
graphdb.auth.oauth.default_roles = ROLE_USER
```

X.509 certificate + local users

Example configuration for X.509 certificate authentication + local user database authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ X.509 AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on X.509 certificate authentication. The default value is 'basic, gdb'.
# Provide only 'x509' if password-based login methods (basic and gdb) will not be used.
graphdb.auth.methods = basic, gdb, x509

# Provide a regular expression to extract the username from the certificate. The default is CN=(.*?)(?:,|
↪$).
# If you want to provide a custom expression, uncomment the below and edit it.
graphdb.auth.methods.x509.subject.dn.pattern = CN=(.*?)(?:,|$)

# To implement the server-side X.509 authentication, enable SSL.
graphdb.connector.ssl.enabled = true

# GraphDB uses the Java implementation of SSL, which requires a configured key in the Java keystore.
# To setup keystore, uncomment the following properties and set 'keystorePass' and 'keyPass' to their
↪actual values.
# The default is the .keystore file in the operating system home directory of the user that is running
↪GraphDB.
graphdb.connector.ssl.certificateKeystoreFile = <path-to-the-keystore-file>
graphdb.connector.ssl.certificateKeystorePassword = <secret>
graphdb.connector.ssl.certificateKeyAlias = graphdb
graphdb.connector.ssl.certificateKeyPassword = <secret>

# (optional) To implement the client-side authentication,
# we need a truststore that holds the certificate authorities (CA) for the intended client certificates.
# To set up a truststore different from the default JRE one,
# uncomment the following properties and set 'truststoreFile' and 'truststorePassword' to their actual
↪values.
graphdb.connector.ssl.truststoreFile = <path-to-custom-truststore-file>
graphdb.connector.ssl.truststorePassword = <secret>

# Configure the X.509 certificate revocation status check. Only one of OCSP and CRLDP can be enabled at a
↪time.
# To enable the check you want, uncomment the other one and set it to false.
# graphdb.auth.methods.x509.ocsp = true
graphdb.auth.methods.x509.crldp = false

# In the event of OCSP or CRLDP responders downtime or certificates that do not provide an Authority
↪Information Access (AIA) extension,
# you can set a Certificate Revocation List (CRL) to Tomcat.
graphdb.auth.methods.x509.crlFile = <path-to-certificate-revocation-list>
```

(continues on next page)

(continued from previous page)

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LOCAL USER AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# The local database is the default setting but it may be set explicitly as such:
graphdb.auth.database = local
```

X.509 certificate + LDAP

Example configuration for X.509 certificate authentication + LDAP authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ X.509 AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on X.509 certificate authentication. The default value is 'basic, gdb'.
# Provide only 'x509' if password-based login methods (basic and gdb) will not be used.
graphdb.auth.methods = basic, gdb, x509

# Provide a regular expression to extract the username from the certificate. The default is CN=(.*?)(?:,|
↪$).
# If you want to provide a custom expression, uncomment the below and edit it.
graphdb.auth.methods.x509.subject.dn.pattern = CN=(.*?)(?:,|$)

# To implement the server-side X.509 authentication, enable SSL.
graphdb.connector.ssl.enabled = true

# GraphDB uses the Java implementation of SSL, which requires a configured key in the Java keystore.
# To setup keystore, uncomment the following properties and set 'keystorePass' and 'keyPass' to their
↪actual values.
# The default is the .keystore file in the operating system home directory of the user that is running
↪GraphDB.
graphdb.connector.ssl.certificateKeystoreFile = <path-to-the-keystore-file>
graphdb.connector.ssl.certificateKeystorePassword = <secret>
graphdb.connector.ssl.certificateKeyAlias = graphdb
graphdb.connector.ssl.certificateKeyPassword = <secret>

# (optional) To implement the client-side authentication,
# we need a truststore that holds the certificate authorities (CA) for the intended client certificates.
# To set up a truststore different from the default JRE one,
# uncomment the following properties and set 'truststoreFile' and 'truststorePassword' to their actual
↪values.
graphdb.connector.ssl.truststoreFile = <path-to-custom-truststore-file>
graphdb.connector.ssl.truststorePassword = <secret>

# Configure the X.509 certificate revocation status check. Only one of OCSP and CRLDP can be enabled at a
↪time.
# To enable the check you want, uncomment the other one and set it to false.
# graphdb.auth.methods.x509.ocsp = true
graphdb.auth.methods.x509.crldp = false

# In the event of OCSP or CRLDP responders downtime or certificates that do not provide an Authority
↪Information Access (AIA) extension,
# you can set a Certificate Revocation List (CRL) to Tomcat.
graphdb.auth.methods.x509.crlFile = <path-to-certificate-revocation-list>

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LDAP AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on ldap authentication and configure the server.
graphdb.auth.module = ldap
graphdb.auth.ldap.url = ldap://localhost:10389/dc=example,dc=org
```

(continues on next page)

(continued from previous page)

```
# Permit access for all users that are part of the "people" unit of the fictional "example.org"
↪organization.
graphdb.auth.ldap.user.search.base = ou=people
graphdb.auth.ldap.user.search.filter = (cn={0})

# Make all users in the Administration group GraphDB administrators as well.
graphdb.auth.ldap.role.search.base = ou=groups
graphdb.auth.ldap.role.search.filter = (member={1})
graphdb.auth.ldap.role.map.administrator = Administration

# Make all users in the Management group GraphDB Repository Managers as well.
graphdb.auth.ldap.role.map.repositoryManager = Management

# Enable all users in the Readers group to read the my_repo repository.
graphdb.auth.ldap.role.map.repository.read.my_repo = Readers

# Enable all users in the Writers group to write and read the my_repo repository.
graphdb.auth.ldap.role.map.repository.write.my_repo = Writers

# Required for accessing a LDAP server that does not allow anonymous binds and anonymous access.
graphdb.auth.ldap.bind.userDn = uid=userId,ou=people,dc=example,dc=org
graphdb.auth.ldap.bind.userDn.password = 123456
```

Kerberos + local users

Example configuration for Kerberos authentication + local user database authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ KERBEROS AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Enable Kerberos authentication and keep Basic and GDB authentication enabled.
graphdb.auth.methods = basic, gdb, kerberos

# Provides the Kerberos keytab file relative to the GraphDB config directory.
graphdb.auth.kerberos.keytab = graphdb-http.keytab

# Provides the Kerberos principal for GraphDB running at data.example.org and Kerberos users from
# the realm EXAMPLE.ORG.
graphdb.auth.kerberos.principal = HTTP/data.example.org@EXAMPLE.ORG

# Enable Kerberos debug messages (recommended when you first setup Kerberos, can be disabled later).
graphdb.auth.kerberos.debug = true

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LOCAL USER AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# The local database is the default setting but it may be set explicitly as such:
graphdb.auth.database = local
```

Kerberos + LDAP

Example configuration for Kerberos authentication + LDAP authorization:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ KERBEROS AUTHENTICATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Enable Kerberos authentication and keep Basic and GDB authentication enabled.
graphdb.auth.methods = basic, gdb, kerberos

# Provides the Kerberos keytab file relative to the GraphDB config directory.
```

(continues on next page)

(continued from previous page)

```
graphdb.auth.kerberos.keytab = graphdb-http.keytab

# Provides the Kerberos principal for GraphDB running at data.example.org and Kerberos users from
# the realm EXAMPLE.ORG.
graphdb.auth.kerberos.principal = HTTP/data.example.org@EXAMPLE.ORG

# Enable Kerberos debug messages (recommended when you first setup Kerberos, can be disabled later).
graphdb.auth.kerberos.debug = true

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LDAP AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on ldap authentication and configure the server.
graphdb.auth.module = ldap
graphdb.auth.ldap.url = ldap://localhost:10389/dc=example,dc=org

# Permit access for all users that are part of the "people" unit of the fictional "example.org"
↪organization.
graphdb.auth.ldap.user.search.base = ou=people
graphdb.auth.ldap.user.search.filter = (cn={0})

# Make all users in the Administration group GraphDB administrators as well.
graphdb.auth.ldap.role.search.base = ou=groups
graphdb.auth.ldap.role.search.filter = (member={1})
graphdb.auth.ldap.role.map.administrator = Administration

# Make all users in the Management group GraphDB Repository Managers as well.
graphdb.auth.ldap.role.map.repositoryManager = Management

# Enable all users in the Readers group to read the my_repo repository.
graphdb.auth.ldap.role.map.repository.read.my_repo = Readers

# Enable all users in the Writers group to write and read the my_repo repository.
graphdb.auth.ldap.role.map.repository.write.my_repo = Writers

# Required for accessing a LDAP server that does not allow anonymous binds and anonymous access.
graphdb.auth.ldap.bind.userDn = uid=userId,ou=people,dc=example,dc=org
graphdb.auth.ldap.bind.userDn.password = 123456
```

6.2.4 Fine-grained access control

By default, permissions in GraphDB are set at the repository level — access to a repository grants access to all its statements indiscriminately. However, a more robust access control system may be necessary in some cases, such as restricting access to a specific named graph (a collection of statements with sensitive data), to a specific predicate (a single statement with sensitive information, such as salary), or restricting users from being able to clear an entire named graph.

Fine-grained access control (FGAC) allows you to manage access to specific resources at a more granular level, so that only authorized users can read or modify certain data, or execute certain operations. The FGAC mechanism provides a means to allow or deny access based on elements such as parts of the RDF statement or specific plugins, as well as to control access to system operations that change the repository in a fundamental way.

Note: To learn how to manage FGAC rules, check the *Managing FGAC rules through Workbench* and *Managing FGAC rules through the REST API* documentation pages.

6.2.4.1 How FGAC works

Fine-grained access control is managed through individual access control rule definitions, with each definition setting the rules for access to a specific resource or operation for a specific group of people. There are four types of rule definitions, based on the scope of the the access control check the rule performs — statement, clear graph, plugin, system. Check *Rule scopes for Fine-grained access control* below for more information on the scopes.

A rule definition consists of four elements:

Policy

There are two permission policies:

- **Allow:** the rule will allow access to the resource
- **Deny:** the rule will deny access to the resource

Note: In real-world applications, it doesn't make sense for a user to be allowed to modify a resource that they cannot read. Thus, some rules are automatically inferred:

- A rule that allows `write` operations also allows `read` operations concerning the same resource.
- A rule that denies `read` operations also forbids `write` operations concerning the same resource.

This approach is based on common sense and practicality.

Custom user roles

The role attribute specifies role membership. It can be one of the following:

- A role name: the user **must have** that role
- `!` followed by a role name: the user **must not have** that role, or in other words, `!` negates the role membership

To learn more about user roles, check our documentation on *Custom user roles*.

Operation types

There are two operation types: `read` and `write`. They are defined only for the statement, plugin and system scope.

- **Read:** any operation that reads something without changing its state, such as read operations of repository statements, queries to plugins, reading the SHACL shapes, or reading the fingerprint.
- **Write:** any operation that writes something and potentially changes its state, such as regular data inserts or deletes, updates handled by plugins, changing the SHACL shapes, or changing of the repository via a system statement.

Tip: The operation type can also be set to `"*"` to explicitly use the rule for both **Read** and **Write** access control checks.

Access control list

All rule definitions are stored in an **access control list** (ACL).

When GraphDB needs to determine whether to grant access to a resource, the rules in the ACL are processed sequentially until either of the following occurs:

- A rule matches the operation type, the role attribute and the scope specific attributes — in that case, GraphDB will follow the rule's policy (either **allow** or **deny** access to the resource).
- The end of the ACL is reached — in this case, the default policy is to **allow** access to the resource.

Thus, the order of your ACL rules is of crucial importance.

All rules share the same *management API*, regardless of their scope.

Tip: Rules are only processed within the scope of the operations they cover. In other words, if a user is initiating an operation handled through a plugin, only the rules within the Plugin scope will be processed.

Note: Duplicate rules (where all elements of the rule definition are identical) are not permitted.

Repository managers and administrators

Any user who has the *repository manager or administrator role* will bypass all FGAC checks and thus be allowed to perform any operation on the repository.

6.2.4.2 Rule scopes for Fine-grained access control

The scope of a FGAC rule definition determines the type of resource the rule applies to (as opposed to the specific resource). A given rule will be used only if its scope matches the access control check that needs to be performed. Scopes may have scope-specific attributes and can be one of the following four types:

- statement scope
- clear graph scope
- plugin scope
- system scope

Each of these is covered in more detail below.

Statement scope

Statement scope FGAC allows you to manage access to specific RDF statements, so that only authorized users can read certain data. This way you can allow or deny access based on any part of an RDF statement, where the full representation of a statement consists of a quad — the subject, the predicate, the object, and the named graph, also known as context. Thus, statement scope access control is also known as quad-based access control.

Common use cases for statement scope fine-grained access control are restricting access to a specific named graph (a collection of statements with sensitive data) or a specific predicate (a single statement with sensitive information, such as a salary).

Note: Access control list support for statement scope was first introduced only for read operations in GraphDB 10.4 under the name "Quad-based access control". Repository statement scope ACL rules are backwards-compatible with GraphDB 10.4 quad-based access control rules. Since quad-based access control rules lacked the scope and operation properties, rules created in GraphDB 10.4 will receive `Statement` scope and `*` operation type upon migrating to GraphDB 10.5 or newer.

In addition, all regular users (i.e., not repository managers or administrators) will be subject to FGAC checks, while in GraphDB 10.4 only users with read access (and not read/write access) to a repository were checked.

Statement scope attributes and special values

The statement scope takes the elements of a quad as additional attributes:

- `subject` — the subject of the RDF statement: an IRI or on RDF-star embedded triple
- `predicate` — the predicate of the RDF statement: an IRI
- `object` — the object of the RDF statement: an IRI, an RDF-star embedded triple or a literal
- `context` — the context of the RDF statement: an IRI

Each of those four elements can take a wildcard value of `*` to match any RDF value when the rule is checked. In addition to that, `context` can also take the special values of `default` for the default graph and `named` for any named graph.

RDF values

All RDF values are specified in N-Triples/Turtle-star format using the full notation, where shortened IRIs and bare numeric or boolean literals are not permitted:

- Valid IRIs — `<http://example.com/data/Person1>`, `<http://www.w3.org/2000/01/rdf-schema#label>`
- Invalid IRIs — `rdf:type` (shortened IRIs not allowed)
- Valid literals — `"My data"`, `"Meine Daten"@de`, `"15"^^<http://www.w3.org/2001/XMLSchema#int>`, `"true"^^<http://www.w3.org/2001/XMLSchema#boolean>`
- Invalid literals — `125`, `78.05`, `true`, `false` (bare literals not allowed)
- Valid RDF-star triples — `<<<http://example.com/data/Person1> <http://www.w3.org/2000/01/rdf-schema#label> "Person 1">>`

Note that it is not possible to specify and match against blank nodes.

See also:

Tutorials on *Protecting entire named graphs* and *Protecting sensitive predicates*.

Clear graph scope

Clear graph scope FGAC control is used to manage who can process operations that delete a set of statements identified only by the named graph or the special case of clearing all graphs.

The clear graph scope FGAC is especially useful for restricting access to the `CLEAR ALL` operation by denying access to it.

Clear graph scope attributes and special values

The clear graph scope takes only `context` as an additional attribute, either as an IRI, a wildcard value of `*` to match any value when the rule is checked, or one of the following special values:

- `default` for the default graph.
- `named` for any named graph.
- `all` for the `CLEAR ALL` operation.

The IRIs are specified in *N-Triples/Turtle-star format using the full notation*, just like for statement scope.

Note: Unlike the other scopes, clear graph does not have operation type.

Implicit CLEAR ALL protection

GraphDB processes `CLEAR ALL` as a bulk operation in GraphDB that is executed without iterating over individual statements. As such, it is not possible to check if the user executing the operation has write access to each statement that would be deleted. In order to prevent inadvertent removal of otherwise protected statements, GraphDB will append a clear graph scope rule denying `CLEAR ALL` if there is at least one rule denying writes to statements or clear graph to any named graph.

See also:

Tutorial on *Protecting against clearing all graphs at once*.

Plugin scope

Plugin scope FGAC is used to manage operations handled through plugins. These operations typically look like regular queries or updates but are answered or executed by a plugin instead of being handled as proper RDF data. You can use plugin scope access control to manage who has access to any specific plugin.

A common use case for the plugin scope FGAC is restricting write operations on GraphDB Connectors to prevent accidental removal of a connector instance.

Plugin scope attributes and special values

The plugin scope takes only `plugin` as an additional attribute, either as a plugin name or a wildcard value of `*` to match any plugin when the rule is checked.

See also:

Tutorial on *Protecting connector instance modifications*.

System scope

System scope FGAC is used to manage the rights to perform any operation involving a system statement. This includes operations such as querying the fingerprint, changing the ruleset, or triggering reinferencing.

A common use case for the system scope FGAC is restricting write operations that change the ruleset of the repository.

Note: It is not possible to specify only a single category of operations (for example, only changing the ruleset).

Note: The system scope plugin does not take any additional attributes and does not have special values.

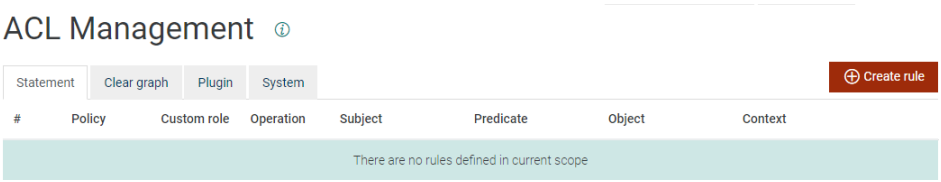
See also:

Tutorial on *Protecting against dangerous system operations*.

6.2.4.3 Managing FGAC rules through Workbench

Note: This documentation relates to *Fine-grained access control*, which requires a *GraphDB Enterprise license*.

Fine-grained access control rules can be easily created and managed through the *Setup* ▸ *ACL Management* view in Workbench. This opens the interface where the ACL rules are loaded and grouped in lists based on their scope. There are four tabs, for the *four different scopes*. While you are not creating a new rule or editing an existing one, you can switch between the different scope tabs freely.

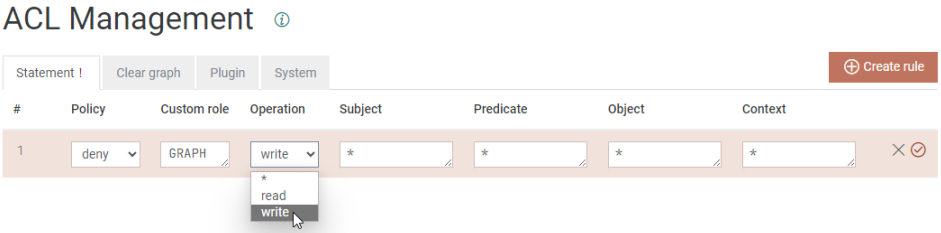


This interface also allows you to create, edit, reorder and delete rules.

Add a new rule

In the tab of the scope you want to create a rule for:

1. Click on the *Create rule* button on the top right of the UI to create a new rule.
2. Enter the parameters you want to apply to your rule.
3. Click on check mark icon to confirm the new rule.



Note:

- New rules are added to the top of the list — you can also add a rule immediately after an existing rule by clicking the plus-shaped icon next to the existing rule.
- You can negate roles (apply rules to all users except the ones with a specific role) by prefixing the role with `!`.
- You can leave any one of the subject, predicate, object, context or plugin (for the plugin scope) as a wildcard placeholder `*` in order to match any value at that position.
- All RDF values are specified in *N-Triples/Turtle-star format using the full notation* and blank nodes are not allowed.
- Custom roles in the UI are specified **without the `CUSTOM_` prefix**, while when *Managing FGAC rules through the REST API* they are specified with the prefix.

Edit an existing rule

In the tab of the scope you want to edit a rule from:

1. Click on edit rule pencil-shaped icon next to the rule you want to edit.
2. Change the parameters you want to modify.
3. Click on the check mark icon to confirm the changes.

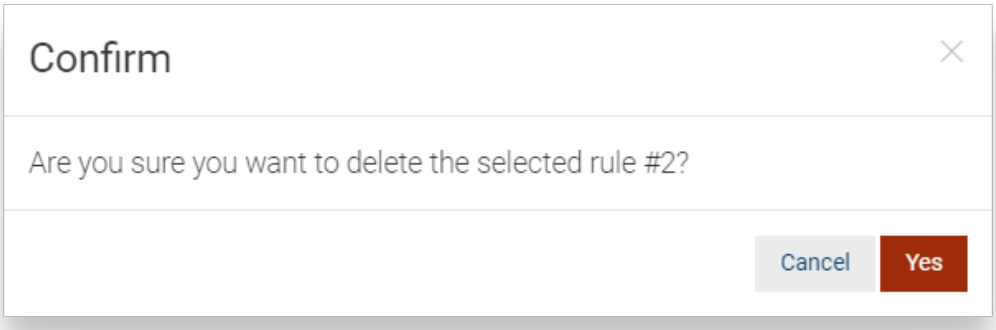
Reordering rules

In the tab of the scope whose rules you want to reorder, click the up and down arrow icons on the left of each rule to move the rule one position up or down respectively.

Deleting an existing rule

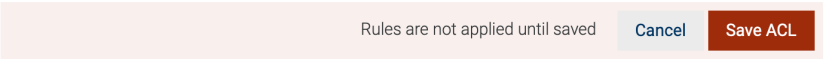
In the tab of the scope you want to delete a rule from:

1. Click on the delete rule trash can icon next to the rule you want to edit.
2. A pop up window will appear asking you to confirm the deletion.



Saving and applying changes

Any changes to the rules — be it adding, deleting, editing or reordering rules — are not preserved nor applied until you click on the *Save ACL* button on the bottom of the page.



> **Warning:** Although each individual fine-grained access control scope has its own management tab in Workbench, all rules are ultimately stored in a single ACL list in the repository.
>
> Because of this, whenever you save changes made to any of the scopes through Workbench, you are saving the changes made to all of the scopes.

Using the UI and the API together

Note that Workbench reorganizes your ACL list by grouping the rules with the same scope together following the order of `statement`, `clear_graph`, `plugin` and `system`. However, this does not change the order of the rules within each scope.

This is important only if you also use the *REST API to manage ACL*. The API will keep the exact same order as provided by the caller when creating rules. Thus, if the rules created via the API are not grouped as described above, they will be reorganized once any rule is edited or created via Workbench.

6.2.4.4 Managing FGAC rules through the REST API

Note: This documentation relates to *Fine-grained access control*, which requires a *GraphDB Enterprise license*.

Fine-grained access control can also be managed through the FGAC management API. Each scope is represented in the API according to a corresponding JSON model:

Statement scope

```
{
    "policy": "allow|deny",
    "role": "role-string",
    "scope": "statement",
    "operation": "read|write|*",
    "subject": "rdf-value|*",
    "predicate": "rdf-value|*",
    "object": "rdf-value|*",
    "context": "rdf-value|named|default|*"
}
```

Clear graph scope

```
{
    "policy": "allow|deny",
    "role": "role-string",
    "scope": "clear_graph",
    "context": "rdf-value|named|default|all|*"
}
```

Plugin scope

```
{
    "policy": "allow|deny",
    "role": "role-string",
    "scope": "plugin",
    "operation": "read|write|*",
    "plugin": "plugin-name-string"
}
```

System scope

```
{
    "policy": "allow|deny",
    "role": "role-string",
    "scope": "system",
    "operation": "read|write|*"
}
```

Note:

- All RDF values are specified in *N-Triples/Turtle-star format using the full notation* and blank nodes are not allowed.
- You can negate roles (apply rules to all users except the ones with a specific role) by prefixing the role with `!`.
- Custom roles specified in JSON **must start with the `CUSTOM_` prefix**, while when *Managing FGAC rules through Workbench* they are specified without the prefix.

The API supports the following operations:

List FGAC rules for a repository

List the *Access control list* for all FGAC rules in the repository. You can use one or more optional request parameters to narrow down the search for specific rules that meet your criteria.

Returns a JSON array of *FGAC rule objects*.

```
GET /rest/repositories/<repositoryID>/acl
```

Optional request parameters for filtering:

- `scope` — The scope of the FGAC rule (for example, `statement`)
- `policy` — The policy for the FGAC rule (`allow` or `deny`)
- `role` — The role associated with the FGAC rule (for example, `CUSTOM_ROLE1` or `!CUSTOM_ROLE1`)
- `operation` — The operation for the FGAC rule (`read`, `write` or `*`)
- `subject` — The subject of the FGAC rule in Turtle-star format (for example, `<http://example.com/Mary>`)
- `predicate` — The predicate of the FGAC rule in Turtle-star format (for example, `<http://www.w3.org/2000/01/rdf-schema#label>`)
- `object` — The object of the FGAC rule in Turtle-star format (for example, `"Mary"@en`)
- `context` — The context of the FGAC rule in Turtle-star format (for example, `<http://example.org/graphs/graph1>`)
- `plugin` — The plugin name for the FGAC rule with plugin scope (for example, `elasticsearch-connector`)

All of these correspond to the individual fields of the FGAC rule object and they must use the same string representation. When a parameter is not provided, the result will not be filtered by that parameter.

Note: If you construct your request manually, pay attention to the required URL-encoding of the request parameters.

Possible response type:

- HTTP Status 200 (OK) — The request was successful, and the response will contain a list of FGAC rules
- HTTP Status 400 (Bad Request) — The request is invalid (i.e. invalid value in any of the filtering parameters)

Add FGAC rules to a repository

Adds new FGAC rules to the repository. Accepts a JSON array of *FGAC rule objects*.

You can also provide an optional URL request parameter **`position`** that specifies the position of the rules to be added. The position is zero-based (0 is the first position). If the position parameter is not provided, the rules are added at the end of the list.

```
POST /rest/repositories/<repositoryID>/acl
```

Possible response type:

- HTTP Status 200 (OK) — The FGAC rules were successfully added
- HTTP Status 400 (Bad Request) — The request is invalid or missing required information (e.g. wrong value or a rule already exists)

Delete FGAC rules from a repository

Deletes the provided FGAC rules from the repository. Accepts a JSON array of *FGAC rule objects*. The provided FGAC rules are removed from the list regardless of their position.

```
DELETE /rest/repositories/<repositoryID>/acl
```

Possible response type:

- HTTP Status 204 (No Content): The FGAC rules were successfully removed if they existed or the rules did not exist and thus nothing was removed
- HTTP Status 400 (Bad Request): The request is invalid or missing required information (e.g. wrong value)

Set FGAC rules of a repository

Replaces the existing FGAC rules of a repository with a new ACL. Accepts a JSON array of *FGAC rule objects*.

```
PUT /rest/repositories/<repositoryID>/acl
```

Possible response type:

- HTTP Status 200 (OK) — The access control list of FGAC rules was successfully updated
- HTTP Status 400 (Bad Request) — The request is invalid or missing required information

Using the API and the UI together

Note that the API will keep the exact same order as provided by the caller when creating rules, while when *using Workbench to manage ACL*, it will reorganize your ACL list by grouping the rules with the same scope together following the order of **`statement`**, **`clear_graph`**, **`plugin`** and **`system`**. However, this does not change the order of the rules within each scope.

This is important only if you create or edit your ACL list via the API and then edit via Workbench.

6.2.4.5 Fine-grained access control tutorials

This section describes several use cases for *Fine-grained access control* (FGAC) and shows you how to use FGAC rules to protect access to various resources. The following knowledge is assumed:

- *Creating repositories*
- *User management* and general *GraphDB access control*
- *Executing SPARQL queries*

Please create and connect to a repository before trying any of the tutorials below, and familiarize yourself with *Managing FGAC rules through Workbench* and *Managing FGAC rules through the REST API*.

Statement scope

Protecting entire named graphs

One of the typical use cases of *FGAC with the statement scope* is protecting entire named graphs. For example, it can be used to isolate the data of multiple sets of users, where each set of users can read and write one or more named graphs without being able to read and write named graphs intended for another set of users.

> **Warning:** Inferred statements are materialized in the virtual named graph `<http://www.ontotext.com/implicit>`. Thus, users can either be granted access to all inferred statements (including statements inferred from data they cannot access) or no inferred statements at all (including statements inferred from data the users can access).

Let's define three projects and assign a named graph to each project as well as a *custom role* that represents access to each project:

- Project 1
  - Named graph: `<http://example.com/project1>`
  - Custom role: `CUSTOM_ALPHA`
- Project 2
  - Named graph: `<http://example.com/project2>`
  - Custom role: `CUSTOM_BETA`
- Project 3
  - Named graph: `<http://example.com/project3>`
  - Custom role: `CUSTOM_GAMMA`

Users can belong to zero or more projects by assigning the corresponding role to them.

Now we can define the business logic:

- Project 1 users must be able to read and write the named graph `<http://example.com/project1>`.
- Project 2 users must be able to read and write the named graph `<http://example.com/project2>`.
- Project 3 users must be able to read and write the named graph `<http://example.com/project3>`.
- Users that do not belong to the corresponding project must not be able to read or write the project named graph.
- None of the project 1/2/3 users have access to any other data, including inferred statements.

We need to also decide what to do with access to statements when a user does not belong to any of the projects. For example:

- Variant 1: Other users cannot access anything
- Variant 2: Other users can access everything
- Variant 3: Other users can access everything but the three named graphs

Let's look at each of these variants and the rules we need.

Variant 1: Other users cannot access anything

This is the simplest variant. First we have rules that allow access to each of the project-specific named graphs followed by a broad deny rule as the last rule in the list. The deny rule restricts both users from project 1/2/3 as well as any other users from accessing statements outside of the project named graphs:

Policy	Custom role	Operation type	Subject	Predicate	Object	Context
allow	`CUSTOM_ALPHA`	*	*	*	*	`<http://example.com/project1>`
allow	`CUSTOM_BETA`	*	*	*	*	`<http://example.com/project2>`
allow	`CUSTOM_GAMMA`	*	*	*	*	`<http://example.com/project3>`
deny	*	*	*	*	*	*

Variant 2: Other users can access everything

In this variant, in addition to the rules that allow access to the three named graphs, we have three additional deny rules that prevent project 1/2/3 users from accessing statements outside their assigned named graph:

Policy	Custom role	Operation type	Subject	Predicate	Object	Context
allow	`CUSTOM_ALPHA`	*	*	*	*	`<http://example.com/project1>`
deny	`CUSTOM_ALPHA`	*	*	*	*	*
allow	`CUSTOM_BETA`	*	*	*	*	`<http://example.com/project2>`
deny	`CUSTOM_BETA`	*	*	*	*	*
allow	`CUSTOM_GAMMA`	*	*	*	*	`<http://example.com/project3>`
deny	`CUSTOM_GAMMA`	*	*	*	*	*

Variant 3: Other users can access everything but the three named graphs

This variant is similar to variant 2 but we changed the `allow` to `deny` and added `!` to negate the role membership:

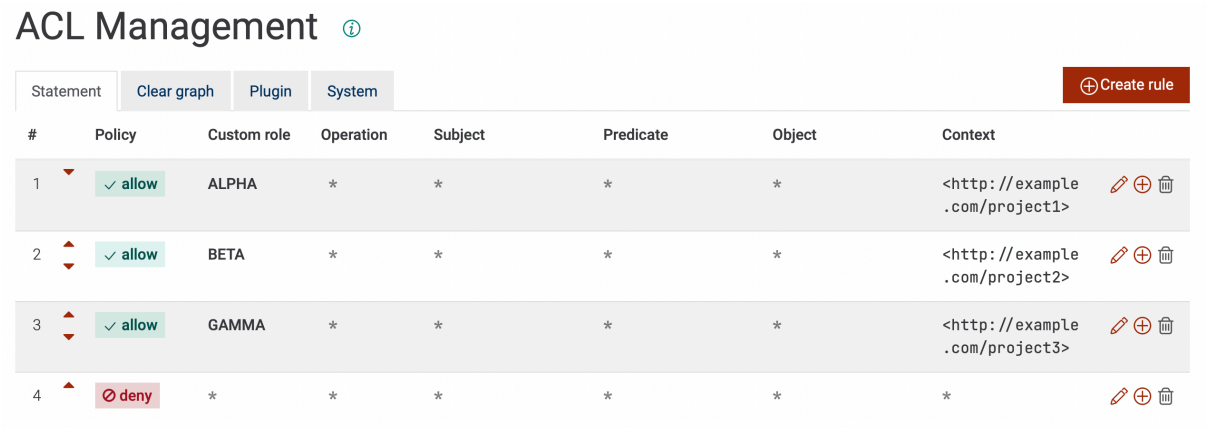
Policy	Custom role	Operation type	Subject	Predicate	Object	Context
deny	`!CUSTOM_ALPHA`	*	*	*	*	`<http://example.com/project1>`
deny	`CUSTOM_ALPHA`	*	*	*	*	*
deny	`!CUSTOM_BETA`	*	*	*	*	`<http://example.com/project2>`
deny	`CUSTOM_BETA`	*	*	*	*	*
deny	`!CUSTOM_GAMMA`	*	*	*	*	`<http://example.com/project3>`
deny	`CUSTOM_GAMMA`	*	*	*	*	*

A rule like `deny !CUSTOM_ALPHA * * * * <http://example.com/project1>` is equivalent to two adjacent rules:

Policy	Custom role	Operation type	Subject	Predicate	Object	Context
allow	`CUSTOM_ALPHA`	*	*	*	*	`<http://example.com/project1>`
deny	*	*	*	*	*	`<http://example.com/project1>`

Putting it together

Let's try Variant 1 in Workbench. The only difference with the rule table above is that custom roles in Workbench are specified without the `CUSTOM_` prefix.



The equivalent REST API call to create the same rules with curl is:

```
curl -u admin:xxx -X POST -H "Content-type: application/json"\
    http://localhost:7200/rest/repositories/test/acl -d '
        [
            {
                "policy": "allow",
                "role": "CUSTOM_ALPHA",
                "scope": "statement",
                "operation": "*",
                "subject": "*",
                "predicate": "*",
                "object": "*",
                "context": "<http://example.com/project1>"
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "policy": "allow",
            "role": "CUSTOM_BETA",
            "scope": "statement",
            "operation": "*",
            "subject": "*",
            "predicate": "*",
            "object": "*",
            "context": "<http://example.com/project2>"
        },
        {
            "policy": "allow",
            "role": "CUSTOM_GAMMA",
            "scope": "statement",
            "operation": "*",
            "subject": "*",
            "predicate": "*",
            "object": "*",
            "context": "<http://example.com/project3>"
        },
        {
            "policy": "deny",
            "role": "*",
            "scope": "statement",
            "operation": "*",
            "subject": "*",
            "predicate": "*",
            "object": "*",
            "context": "*"
        }
    ]'
```

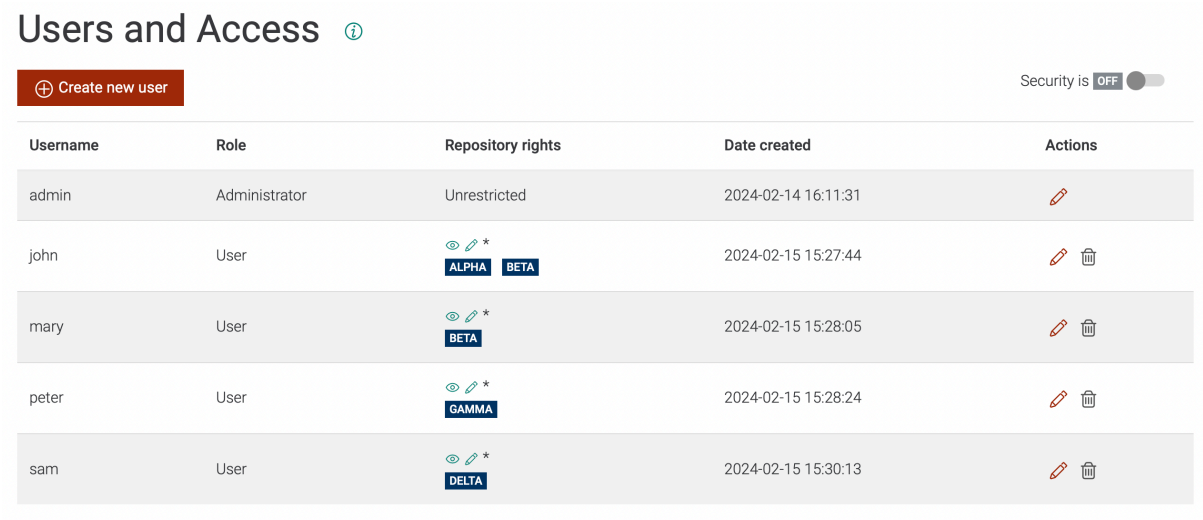
The above curl call will add the rules to the end of the existing list so it assumes the existing list is empty. The JSON definition of the rules is very similar to the rules presented in the table above but here we need to also specify the relevant scope (statement). Unlike in Workbench, the custom roles are specified fully, including the `CUSTOM_` prefix.

Note: Note that you need to replace the `xxx` placeholder with your admin user password.

We also need to create some users that belong to zero or more projects so we can test our rules:

- John — belongs to project 1 (roles `CUSTOM_ALPHA` and `CUSTOM_BETA`)
- Mary — belongs to project 2 (role `CUSTOM_BETA`)
- Peter — belongs to project 3 (role `CUSTOM_GAMMA`)
- Sam — does not belong to any project but has the role `CUSTOM_DELTA`

The users need to have read/write access to the test repository.



First, log in as admin (to bypass all ACL checks) and execute this update to provide some test data:

```
PREFIX ex: <http://example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    GRAPH ex:project1 {
        ex:Project1 a ex:Project .
    }
    GRAPH ex:project2 {
        ex:Project2 a ex:Project .
    }
    GRAPH ex:project3 {
        ex:Project3 a ex:Project .
    }
}
```

If we now log in as the user John we can test the following:

- Insert/delete data in the named graphs `<http://example.com/project1>` and `<http://example.com/project2>` is allowed:

```
PREFIX ex: <http://example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    GRAPH ex:project1 {
        ex:John rdfs:label "John works on Project 1" .
    }
    GRAPH ex:project2 {
        ex:John rdfs:label "John works on Project 2 too" .
    }
}
```

- Reading data in the same named graphs is allowed:

```
SELECT * WHERE {
    ?s ?p ?o
}
```

This query should return all the statements from the two named graphs:

s	p	o
ex:Project1	rdf:type	ex:Project
ex:Project2	rdf:type	ex:Project
ex:John	rdfs:label	"John works on Project 1"
ex:John	rdfs:label	"John works on Project 2 too"

- Insert/delete data outside the two named graphs is denied with an error message:

```
PREFIX ex: <http://example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    GRAPH ex:project3 {
        ex:John rdfs:label "Does John work on Project 3?" .
    }
}
```

  This update will return the error *No write permission to statement: <http://example.com/John> <http://www.w3.org/2000/01/rdf-schema#label> "Does John work on Project 3?" <http://example.com/project3>*.

- Any statements outside the two named graphs are not visible.

  See the SELECT query above. If John could see other statements, the statement `ex:Project3 rdf:type ex:Project` would be in the results too.

Similarly, if we log in as Mary or Peter we should be able to read and write statements in the respective named graphs `<http://example.com/project2>` and `<http://example.com/project3>`.

If we log in as Sam (who does not belong to any of the three projects), we should not be able to see any statements, nor write any data in the repository.

Protecting sensitive predicates

Another typical use case for *statement-scoped FGAC* is the necessity to restrict access to statements that have a common predicate leading to sensitive data. For example, this could be the salary for a given position at your company.

First, let's insert some test data where salaries are specified via the `ex:salary` predicate. While logged in as admin, execute the following in the SPARQL editor:

```
PREFIX ex: <http://example.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

INSERT DATA {
    ex:George a ex:Employee ;
        rdfs:label "George O. Price" ;
        ex:salary 120000 .

    ex:Teresa a ex:Employee ;
        rdfs:label "Teresa M. Phillips" ;
        ex:salary 140000 .
}
```

We can broadly define the access requirements to salary statements such that:

- Payroll needs to be able to read those statements. Let's assign the *custom role* `CUSTOM_PAYROLL` to those users.
- Management needs to be able to read and write salary statements. Let's assign the custom role `CUSTOM_MANAGEMENT` to those users.

- Everyone else must not see any salary statements or be able write any such statements. No need to assign any custom role.

Thus, the rules to achieve the above business logic look like this:

Policy	Custom role	Operation type	Subject	Predicate	Object	Context
allow	`CUSTOM_PAYROLL`	read	*	<http://example.com/salary>	*	*
deny	`!CUSTOM_MANAGEMENT`	*	*	<http://example.com/salary>	*	*

or the more verbose equivalent:

Policy	Custom role	Operation type	Subject	Predicate	Object	Context
allow	`CUSTOM_PAYROLL`	read	*	<http://example.com/salary>	*	*
allow	`CUSTOM_MANAGEMENT`	*	*	<http://example.com/salary>	*	*
deny	*	*	*	<http://example.com/salary>	*	*

Let's use the simpler variant and define those rules in Workbench:

Alternatively, add the same rules via curl:

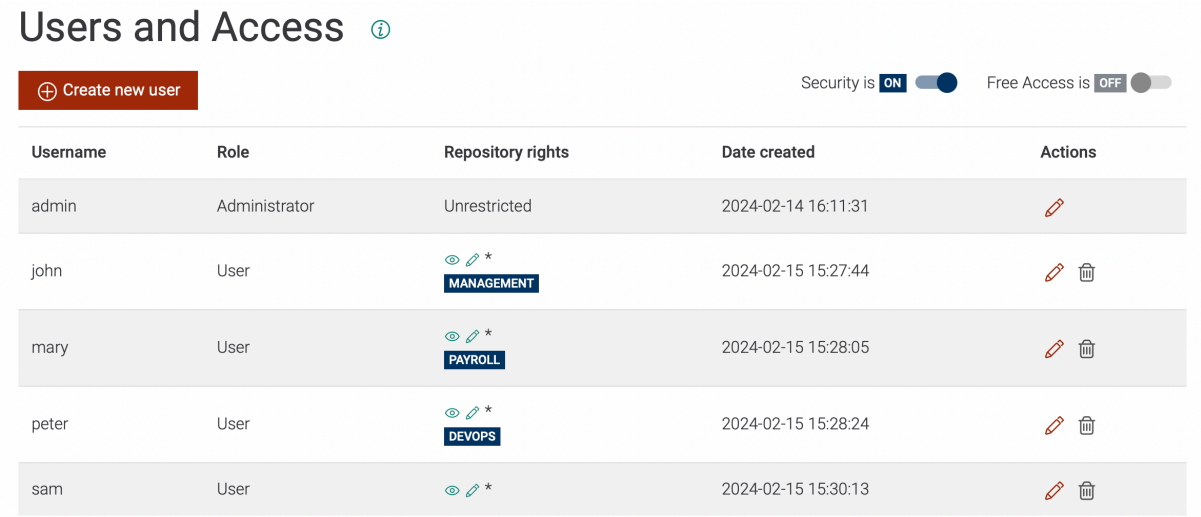
```
curl -u admin:xxx -X POST -H "Content-type: application/json"\
    http://localhost:7200/rest/repositories/test/acl -d '
        [
            {
                "policy": "allow",
                "role": "CUSTOM_PAYROLL",
                "scope": "statement",
                "operation": "read",
                "subject": "*",
                "predicate": "<http://example.com/salary>",
                "object": "*",
                "context": "*"
            },
            {
                "policy": "deny",
                "role": "!CUSTOM_MANAGEMENT",
                "scope": "statement",
                "operation": "*",
                "subject": "*",
                "predicate": "<http://example.com/salary>",
                "object": "*",
                "context": "*"
            }
        ]'
```

Let's create some users that belong to our custom roles:

- John — belongs to Management (role `CUSTOM_MANAGEMENT`)

- Mary — belongs to Payroll (role `CUSTOM_PAYROLL`)
- Peter — a user that has an unrelated custom role `CUSTOM_DEVOPS`
- Sam — a user that has no custom roles

Note: If you already have these user accounts from early exercises on this page, delete the custom roles added for the purposes of those exercises before adding these new roles. Also, delete any leftover rules from those exercises on the ACL Management screen.



Now, if we log in as John we should be able to query salaries:

```
PREFIX ex: <http://example.com/>

SELECT * WHERE {
    ?employee a ex:Employee ;
        ?p ?o .
}
```

employee	p	o
ex:George	rdf:type	ex:Employee
ex:George	rdfs:label	“George O. Price”
ex:George	**ex:salary**	**“120000”^^xsd:integer**
ex:Teresa	rdf:type	ex:Employee
ex:Teresa	rdfs:label	“Teresa M. Phillips”
ex:Teresa	**ex:salary**	**“140000”^^xsd:integer**

Since John is in Management, he is allowed to change the salaries. Let’s give Teresa a raise:

```
PREFIX ex: <http://example.com/>

DELETE {
    ex:Teresa ex:salary ?salary
} INSERT {
    ex:Teresa ex:salary 150000
} WHERE {
    ex:Teresa ex:salary ?salary
}
```

If you now log in as Mary you will still be able to query the salaries since Mary is part of Payroll. If you attempt to change a salary, you will receive an error: *No write permission to statement: <http://example.com/Teresa> <http://example.com/salary> “140000”^^<http://www.w3.org/2001/XMLSchema#integer> <default graph>.*

Similarly, if you log in as Peter or Sam, you will not see any salary statements, nor be able to add or delete them. You will still be able to see and modify all non-salary statements.

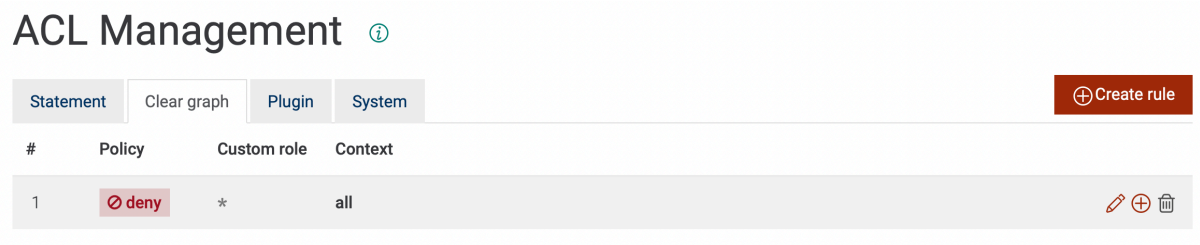
Clear graph scope

SPARQL `CLEAR` and equivalent operations in the RDF4J Java and REST APIs are a convenient way to delete a set of statements without specifying each statement individually. This can be very convenient when you want to replace the statements with new statements but it is a dangerous operation that can be executed by mistake.

Protecting against clearing all graphs at once

`CLEAR ALL` (clearing all graphs in the repository, including all named graphs and the default graph) is a particularly dangerous operation that has little use in day-to-day workflows. We can use the *clear graph scope* to define a rule that prevents users from executing `CLEAR ALL`. The special value `all` in the context position of the rule will match `CLEAR ALL` and equivalent operations.

In Workbench:



Or add the same rule with curl:

```
curl -u admin:xxx -X POST -H "Content-type: application/json"\
    http://localhost:7200/rest/repositories/test/acl -d '
        [
            {
                "policy": "deny",
                "role": "*",
                "scope": "clear_graph",
                "context": "all"
            }
        ]'
```

Attempting to execute CLEAR ALL will produce an error: *No clear graph permission to: all graphs*.

Tip: You will still be able to execute `CLEAR ALL` with any user that is at least a repository manager. You can also assign a *custom role* (for example, `CUSTOM_POWERUSER`) to trusted regular users and change the rule like this:

Policy	Custom role	Context
deny	`!CUSTOM_POWERUSER`	all

Plugin scope

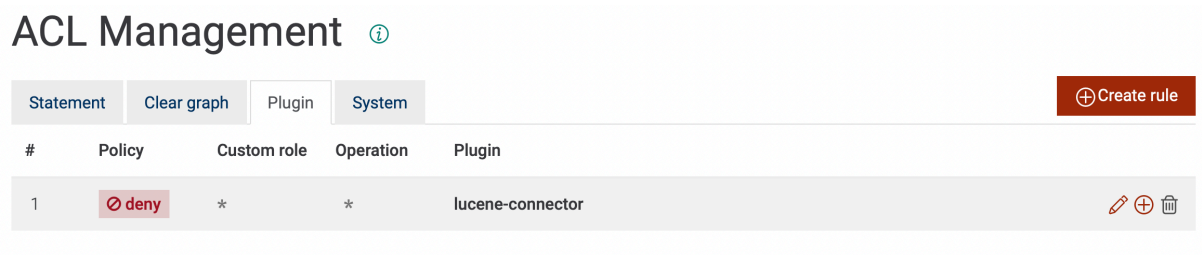
Some queries and updates in GraphDB are handled by plugins. A typical example is the *GraphDB Connectors*, where creating a connector instance involves a SPARQL update that is executed by the respective plugin, and querying a connector instance is handled by the same plugin.

Protecting connector instance modifications

Plugins like the Connectors can play a central role in your project. Once a Connector instance is created, it is typically only used for queries. However, any users with write access to the repository can delete the Connector instance or recreate it with different settings. We can use the *plugin scope* to create rules that prevent this. For example, to protect *Lucene GraphDB Connector* instances you need a deny rule that restricts write access to the `lucene-connector` plugin.

Tip: You can see a list of all plugins and their names in the *Setup* ▸ *Plugins* view of Workbench.

In Workbench:



Or add the same rule with curl:

```
curl -u admin:xxx -X POST -H "Content-type: application/json"\
    http://localhost:7200/rest/repositories/test/acl -d '
        [
            {
                "policy": "deny",
                "role": "*",
                "scope": "plugin",
                "operation": "*",
                "plugin": "lucene-connector"
            }
        ]'
```

Attempting to *create* or *drop* a Lucene Connector instance will produce an error: *No write permission to plugin: lucene-connector*.

Tip: You will still be able to write to the lucene-connector plugin with any user that is at least a repository manager. You can also assign a *custom role* (for example, `CUSTOM_POWERUSER`) to trusted regular users and change the rule like this:

Policy	Custom role	Operation	Plugin
deny	`!CUSTOM_POWERUSER`	write	`lucene-connector`

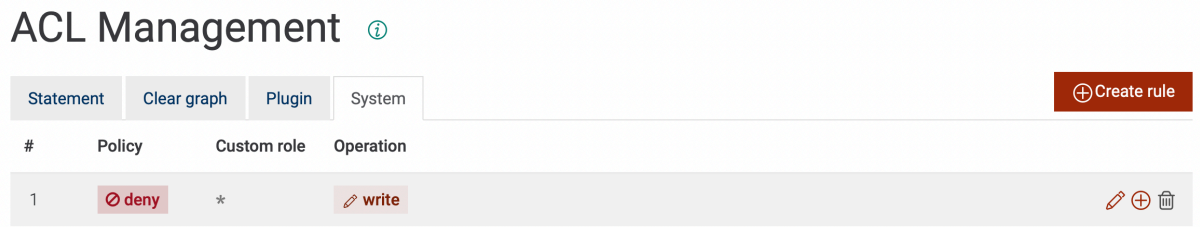
System scope

Some queries and updates in GraphDB are neither regular data nor handled by plugins. These typically use system predicates that allow users to query the fingerprint, the current ruleset, change the ruleset, trigger reinfer, etc.

Protecting against dangerous system operations

While queries to system predicates are generally safe (they reveal some system information about the repository), updates via system predicates can be used to change the ruleset and thus directly affect the statements stored in the repository. If you want to prevent users from changing the state of the repository via system operations you can define a deny rule in the system scope for write operations.

In Workbench:



Or add the same rule with curl:

```
curl -u admin:xxx -X POST -H "Content-type: application/json"\
    http://localhost:7200/rest/repositories/test/acl -d '
        [
            {
                "policy": "deny",
                "role": "*",
                "scope": "system",
                "operation": "write"
            }
        ]'
```

Attempting to execute an update with a system predicate will produce an error.

For example, adding a ruleset:

```
PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA {
  _:b sys:addRuleset "owl-horst-optimized" .
  _:b sys:defaultRuleset "owl-horst-optimized" .
}
```

Executing the update will return an error message: *No write permission to system operation: addRuleset*.

Tip: You will still be able to execute system updates with any user that is at least a repository manager. You can also assign a *custom role* (for example, `CUSTOM_POWERUSER`) to trusted regular users and change the rule like this:

Policy	Custom role	Operation
deny	`!CUSTOM_POWERUSER`	write

6.2.5 Custom user roles

Custom user roles are a key component of managing security through the *Fine-grained access control* and *Role-Based Access Control*. In order to allow or deny access using fine-grained access control, you need to grant custom roles to users according to your needs.

Custom roles are similar to the *GraphDB system user roles*. However, unlike those, custom roles have no inherent rights or restrictions associated with them, and are instead just markers that can be freely granted to any number of users. You can also grant each user any number of custom roles. Thus, custom roles provide the facility for grouping users together for the sake of defining fine-grained access control rules.

Custom roles are identified by the prefix `CUSTOM_` in their name. For example, `CUSTOM_ROLE_ADMIN` and `CUSTOM_ADMIN` are custom roles, while `ROLE_ADMIN` (one of the system roles) and `ADMIN_ROLE` (neither a system role, nor a custom one) are not custom roles.

The names of custom roles are not case sensitive — `CUSTOM_ROLE_ADMIN`, `custom_role_admin` and `Custom_Role_ADMIN` all refer to the same custom role — but they will be normalized to uppercase when returned by API calls and shown in the user interface.

6.2.5.1 Mapping OAuth custom roles

The `graphdb.auth.oauth.bare_custom_roles` property gives you greater control over how custom roles in OAuth tokens map to GraphDB. When this property is set to `false`, custom roles in the OAuth tokens need the `CUSTOM_` prefix in order to be recognized as such. For example, `CUSTOM_FOO` in a token would map to `CUSTOM_FOO` in GraphDB.

When `graphdb.auth.oauth.bare_custom_roles` is set to to `true`, custom roles in an OAuth token do not need the `CUSTOM_` prefix. For example, `FOO` in an OAuth token would map to `CUSTOM_FOO` in GraphDB. You can still use the prefix, so in addition to that `FOO` role, you may also have `CUSTOM_BAR` in an OAuth token, which will map to `CUSTOM_BAR` in GraphDB.

For more information on how to control this behavior, see the *OAuth user database* documentation.

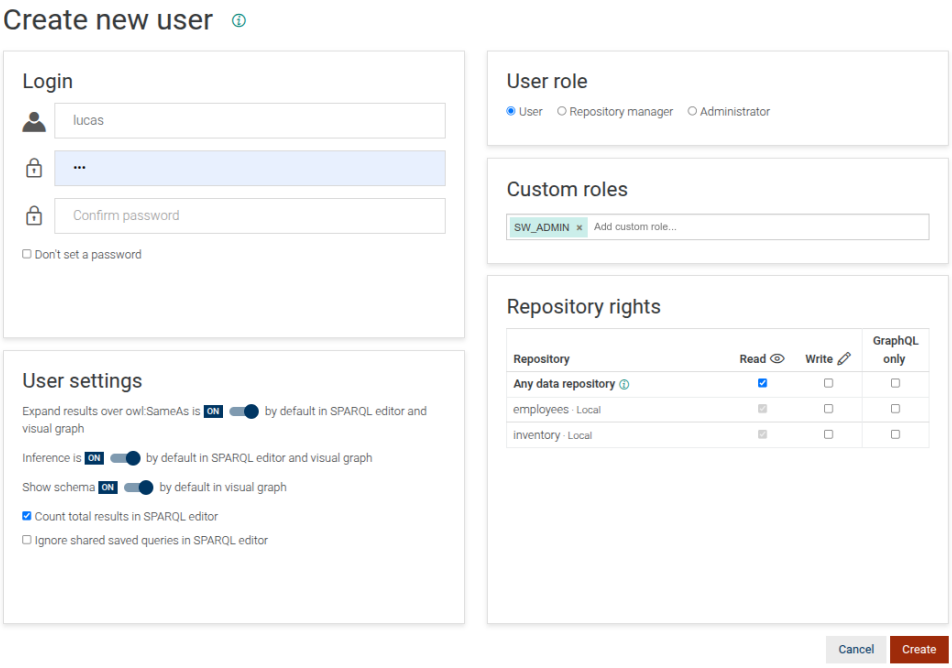
6.2.5.2 Managing custom roles through Workbench

Granting or revoking custom roles can be done easily through the Workbench UI:

1. Navigate to to *Setup* ▸ *Users and Access*.
2. Start *creating a new user or editing an existing one*.

To grant a custom role, simply input the name of the custom role without the `CUSTOM_` prefix in the *Custom Roles* field.

To revoke a custom role, click the x icon of the custom role you want to revoke.



Note: The `CUSTOM_` prefix is omitted in the User management view in Workbench for convenience.

Note: The Workbench UI allows you to grant custom roles only to basic *Users*. *Admin* and *Repository manager* users are automatically granted access to all statements and thus assigning custom roles to them has no practical use. However, if you add custom roles to those users through *the REST API*, they will also be displayed in Workbench.

6.2.5.3 Custom role management REST API

Custom roles can also be managed through the custom role management REST API, which provides additional capabilities.

Users in a role can be represented as a simple JSON array in the API:

```
[
  "user1",
  "user2"
]
```

Conversely, custom roles are described by a simple JSON object where the key is the role name and the value is the user array:

```
{
  "custom_role1": [
    "user1",
    "user2"
  ],
  "custom_role2": [
    "user1"
  ],
  "custom_role3": [
    "user2",
    "user3"
  ]
}
```

The API supports the following operations:

List all custom roles and their users

Returns a *JSON custom roles object* that contains all custom roles and the users associated with each role.

```
GET /rest/security/custom-roles
```

Possible response type:

- HTTP Status 200 (OK) — The request was successful, and the response will contain a *JSON custom roles object*

Set all custom roles and their users

Replaces all existing custom roles/users associations with the provided ones. Accepts a *JSON custom roles object*.

```
PUT /rest/security/custom-roles
```

Possible response type:

- HTTP Status 200 (Created) — The custom roles were successfully updated
- HTTP Status 400 (Bad Request) — The request is invalid (e.g. one of the provided users does not exist, or one of the provided roles is not a custom role)

List the users of a particular custom role

Returns a JSON array of usernames as strings that contains all users associated with the provided custom role.

```
GET /rest/security/custom-roles/<customRole>
```

Possible response type:

- HTTP Status 200 (OK) — The request was successful, and the response will contain an array of usernames as strings
- HTTP Status 400 (Bad Request) — The request is invalid (e.g. the provided role is not a custom role)

Set custom role users

Replaces all users in `<customRole>` with the provided ones. Accepts a JSON array of usernames as strings.

```
PUT /rest/security/custom-roles/<customRole>
```

Possible response type:

- HTTP Status 200 (OK) — The custom role was successfully updated
- HTTP Status 400 (Bad Request) — The request is invalid (e.g. one of the provided users does not exist, or the provided role is not a custom role)

Grant custom role to user

Adds the provided users to `<customRole>`. Accepts a JSON array of usernames as strings.

```
POST /rest/security/custom-roles/<customRole>
```

Possible response type:

- HTTP Status 200 (OK) — The custom role was granted to each of the provided users who was not already granted that role
- HTTP Status 400 (Bad Request) — The request is invalid (e.g. one of the provided users does not exist, or the provided role is not a custom role)

Revoke custom role from user

Removes the provided users from `<customRole>`. Accepts a JSON array of usernames as strings.

```
DELETE /rest/security/custom-roles/<customRole>
```

Possible response type:

- HTTP Status 204 (No Content) — The custom role was revoked if it was previously granted or no action was taken if the role was not already granted
- HTTP Status 400 (Bad Request) — The request is invalid (e.g. one of the provided users does not exist, or the provided role is not a custom role)

List all roles associated with a user

Returns a JSON array of roles as strings with the custom roles granted to the provided user.

```
GET /rest/security/users/<username>/custom-roles
```

Possible response type:

- HTTP Status 200 (OK) — The request was successful, and the response will contain a list of custom roles

6.2.6 Encryption

6.2.6.1 Encryption in transit

All network traffic between the clients and GraphDB and between the different GraphDB nodes (in case of a cluster topology) can be performed over either HTTP or HTTPS protocols. It is highly advisable to encrypt the traffic with SSL/TLS because it has numerous security benefits.

Configuring a GraphDB instance with SSL

As GraphDB runs on embedded Tomcat server, the security configuration is standard with a few exceptions. See more in the official Tomcat documentation on how to enable SSL/TLS.

SSL can be enabled by configuring the following parameter:

```
graphdb.connector.ssl.enabled = true
```

GraphDB uses the Java implementation of SSL, which requires a configured key in the Java keystore.

```
graphdb.connector.ssl.certificateKeystoreFile = <path to the keystore file>
graphdb.connector.ssl.certificateKeystorePassword = <secret>
graphdb.connector.ssl.certificateKeyAlias = graphdb
graphdb.connector.ssl.certificateKeyPassword = <secret>
```

If you have no Java keystore, you can generate one by using one of the following methods:

Option one — generate a self-signed key. You would have to trust the certificate in all clients, including all nodes that run in a different JVM.

```
$ keytool -genkey -alias graphdb -keyalg RSA -keystore \path\to\my\keystore
```

Option two — convert a third party trusted OpenSSL certificate to `PKCS12` key and then import to the Java keystore.

```
openssl pkcs12 -export -in mycert.crt -inkey mykey.key
               -out mycert.p12 -name tomcat -CAfile myCA.crt
               -caname root -chain
```

For any additional encryption information, please refer to the *Encryption* section or, since GraphDB runs in an embedded Tomcat, to the Tomcat ssl documentation.

In addition to the above settings, you can set any Tomcat Connector attribute through a property:

```
graphdb.connector.<attribute> = xxx
```

Currently, GraphDB does not support configuration of the SSLHostConfig part of the Tomcat configuration. So when configuring SSL, please refer to the Connector attributes and not the SSLHostConfig ones. See the Tomcat attributes documentation for more information.

Certificate trust

After configuring the GraphDB instance with SSL, certificate trust should be set up between the GraphDB node and all client nodes communicating with it. Certificate trust can be provided in one of two ways:

Use certificates signed by a trusted certification authority

This way, you will not need any additional configuration and the clients will not get security warning when connecting to the server. The drawback is that these certificates are usually not free and you need to work with a third-party CA. We will not look at this option in more detail as creating such a certificate is highly dependent on the CA.

Use self-signed certificates

The benefit is that you generate these certificates yourself and they do not need to be signed by anyone else. However, the drawback is that by default, the nodes will not trust each others' certificates.

If you generate a separate self-signed certificate for each node in the communication, this certificate would have to be present in the Java Truststores of all other nodes. You can do this by either adding the certificate to the default Java Truststore or specifying an additional Truststore when running GraphDB. Information on how to generate a certificate, add it to a Truststore, and make the JVM use this Truststore can be found in the official Java documentation.

However, this method introduces a lot of configuration overhead. Therefore, we recommend that instead of separate certificates for each node, you generate a single self-signed certificate and use it on all nodes. GraphDB extends the standard Java TrustManager, so it will automatically trust its own certificate. This means that if all nodes involved in the communication are using a shared certificate, there would be no need to add it to the Truststore.

Another difference from the standard Java TrustManager is that GraphDB has the option to disregard the hostname when validating the certificates. If this option is disabled, it is recommended to add all possible IPs and DNS names of all nodes that will be using the certificate as Subject Alternative Names when generating the certificate (wildcards can be used as well).

Both options for trusting your own certificate and skipping the hostname validation are configurable from the `graphdb.properties` file:

- `graphdb.http.client.ssl.ignore.hostname` — `false` by default
- `graphdb.http.client.ssl.trust.own.certificate` — `true` by default

6.2.6.2 Encryption at rest

GraphDB does not provide encryption for its data. All indexes and entities are stored in binary format on the hard drive. It should be noted that the data from them can be easily extracted in case somebody gains access to the data directory.

This is why it is recommended to implement some kind of disk encryption on your GraphDB server. There are multiple third-party solutions that can be used.

GraphDB has been tested on LUKS-encrypted hard drive, and no noticeable performance impact has been observed. However, please keep in mind that such may be present, as it is highly dependent on your specific use case.

6.2.7 Security auditing

Audit trail enables accountability for actions. The common use cases are to detect unauthorized access to the system, trace changes to the configuration, and prevent inappropriate actions through accountability.

You can enable the detailed audit trail log by using the `graphdb.audit.role` configuration parameter. Here is an example:

```
graphdb.audit.role=USER
```

The hierarchy of audit roles is as follows:

1. ANY
2. USER
3. REPO_MANAGER
4. ADMIN
5. Logging form (always logged!)

In addition, logging for repository access can be configured by using the `graphdb.audit.repository` property. For example:

```
graphdb.audit.repository=WRITE
```

will lead to all write operations being logged. Read permissions also include write operations.

The detail of the audit trail increases depending on the role that is configured. For example, configuring the audit role for REPO_MANAGER means that access to the repository management resources will be logged, as well as access to the administration resources and the logging form. Configuring the audit role to ADMIN will only log access to the administration resources and the logging form.

The ANY role lists all requests towards resources where that requires authentication.

The following fields are logged for every successful security check:

- Username

- Source IP address
- Response status code
- Type of request method
- Request endpoint
- X-GraphDB-Repository header value or, if missing, which repository is being accessed
- Serialization of the request headers specified in the `graphdb.audit.headers` parameter
- Serialization of all input HTTP parameters and the message body, limited by the `graphdb.audit.request.max.length` parameter

By default, no headers are logged. The `graphdb.audit.headers` parameter configuring this can take multiple values. For instance, if you want to log two headers, you will simply list them with commas:

```
graphdb.audit.headers = Referer,User-Agent
```

The amount of bytes from the message body which get logged defaults to 1,024 if the `graphdb.audit.request.max.length` parameter is not set.

Note: Logs can be space-intensive, especially if you toggle them to level 1 or 2 as described above.

You can configure GraphDB **security settings** and **user profiles and rights** from the Workbench under *Setup ▸ Users and Access*.

For **access control**, GraphDB implements Spring Security. When an HTTP request is received, Spring Security intercepts it, verifies the permissions, and either grants or denies access to the requested database resource or API.

GraphDB supports three types of user databases used for authentication and authorization: Local, LDAP, and OAuth. Each of them contains and manages the user information. GraphDB supports four authentication methods: Basic, GDB, OpenID and Kerberos. Each authentication method is responsible for a specific type of credentials or tokens.

GraphDB supports **encryption in transit** with SSL/TLS certificates for encrypting the network traffic between the clients and GraphDB, and between the different GraphDB nodes (when in a cluster).

GraphDB's detailed **security audit trail** provides actions accountability, and is hierarchically structured in audit roles. The level of detail of the audit depends on the role that is configured.

6.3 Backup and restore

GraphDB supports the backup and restore of both a single GraphDB instance and a *cluster* through its recovery REST API. Both partial (per-repository) and full recovery procedures are available with optional inclusion of user account data. Backups can be created both locally and on *select cloud offerings*.

Note: As with all operations that involve a REST API, in order to perform a backup or a restore procedure:

- The respective GraphDB instance must be online.
- The cluster must be writable — in other words, the majority of its nodes must be active.

Starting with version 10.4.0, GraphDB uses LZ4 compression for backup and restore. Parallel backup compression and parallel S3 streaming are only available with licenses with four or more licensed cores.

Warning: Compressed backups created with GraphDB 10.4.0 and newer are not backward compatible and cannot be restored with older versions of GraphDB. However, restore procedures *are* backward compatible — in other words, in GraphDB 10.4.0 and newer you are able to restore backups created with older versions of GraphDB.

6.3.1 Planning a backup

Whether you want to be able to quickly recover your data in case of failure or perform routine admin operations such as upgrading a GraphDB instance, it is important to prepare an optimal backup & restore procedure.

There are various factors to take into consideration when designing a backup strategy, such as:

- Optimal timing for downtime tolerance for applying backup
- Read-only tolerance on a single node setup for creating a backup
- Load-balanced backup creation (backup is created by one of the followers, so if a quorum exists, updates will be processed)
- Scope of the backed-up data (for example, full or per-repository backup, or whether user accounts and settings are included)
- Available system resources and specifically ensuring enough disk space for backup
- Frequency of backup creation

Note: Only administrators are able to create and restore backups.

It's worth keeping in mind that creating and restoring a backup will change the read and write behavior of GraphDB. These changes are affected by the type of operation (read or write), the type of backup (full or partial), and the deployment (a single GraphDB instance or a cluster).

Based on those criteria, an operation can be affected in one of three ways:

- ✓ Allowed (not affected by the backup or restore)
- × Rejected
- ⌛ Waiting (it will be processed once the backup or restore procedure has been completed)

Backup							
Backup type	**Deployment**	**Write**	**Read**	**Write in affected repository**	**Write in unaffected repository**	**Read in affected repository**	**Read in unaffected repository**
Partial	Cluster	N/A	✓	✓	✓	N/A	N/A
	Single node	N/A	✓	×	✓	N/A	N/A
Full	Cluster	✓	✓	N/A	N/A	N/A	N/A
	Single node	×	✓	N/A	N/A	N/A	N/A

Restore							
Restore type	**Deployment**	**Write**	**Read**	**Write in affected repository**	**Write in unaffected repository**	**Read in affected repository**	**Read in unaffected repository**
Partial	Cluster	N/A	N/A	⌛	⌛	×	✓
	Single node	N/A	N/A	⌛	✓	×	✓
Full	Cluster	×	×	N/A	N/A	N/A	N/A
	Single node	×	×	N/A	N/A	N/A	N/A

Note: For the purposes of this table, `restoreSystemData` should be considered as full restore since it also affects user data, which is not repository-specific.

6.3.1.1 Planning cloud backups

You can also create a backup saved in the cloud, and restore from backup stored on cloud storage. The supported cloud storage options are Amazon S3 and S3-compatible services, Azure Blob, and Google Cloud Storage.

Cloud backup and restore have the same options as regular *GraphDB backup* and regular *GraphDB restore*, with an additional `bucketUri` parameter that contains all the information about the cloud bucket.

Creating an Amazon S3 or compatible services bucket

Amazon S3

For Amazon's S3, the `bucketUri` parameter uses the following format:

```
s3:///<bucket-name>/<backup-name>?region=<AWSRegion>&AWS_ACCESS_KEY_ID=<key-id>&AWS_SECRET_ACCESS_KEY=
↪<access-key>
```

S3 compatible services

For S3 compatible services, the `bucketUri` parameter uses the following format:

```
s3://[<endpoint-hostname>:<endpoint-port>]/<bucket-name>/<backup-name>?region=<AWSRegion>&AWS_ACCESS_
↪KEY_ID=<key-id>&AWS_SECRET_ACCESS_KEY=<access-key>
```

The `endpoint-hostname` and `endpoint-port` values are only used for S3 compatible services.

If either `AWS_ACCESS_KEY_ID` or `AWS_SECRET_ACCESS_KEY` isn't provided, it will fall back to the AWS standardized credential providers.

If `region` is not provided, it will attempt to use the default AWS region provider chain to resolve the configured region. If that fails, it will default to `us-east-1` — US East (N. Virginia). Furthermore, if the provided region does not match the one where the bucket is located, once a connection to the bucket has been established, all subsequent requests will automatically be directed to the region the bucket resides in.

In addition to the default backup options, you can also set up these global parameters in your `graphdb.properties` file before starting your GraphDB instance:

Property name	Description	Default value
`graphdb.s3.tls.enabled`	Enables TLS for secure connection to S3 compatible services	`false`
`graphdb.s3.backup.httpclient.write.timeout`	Timeout in seconds for a cloud backup's single part upload	`3600`
`graphdb.s3.backup.disable.checksum` **Warning:** Set to `true` only if the S3 compatible service you are using doesn't support trailing checksums.	Disables the S3 checksum validation when uploading backups.	`false`

Creating an Azure Blob bucket

For Azure Blob, the `bucketUri` parameter uses the following format:

```
az://<container-name>/<backup-name>?blob_storage_account=<storage_account_name>
```

The `container-name`, `backup-name` and `storage_account_name` values are required.

For shared account key, the `bucketUri` parameter uses the following format:

```
az://<container-name>/<backup-name>?blob_storage_account=<storage_account_name>&blob_access_key=<SAKey>
```

The SAS token is already in Uri format, which is why it is appended after the storage account.

Warning: Some SAS token generators include a `?` in front of the SAS token. The `?` is a delimiter character — it is not a part of the SAS token and should not be included.

Example token usage

```
az://<container-name>/<backup-name>?blob_storage_account=<storage_account_name>&sv=2022-11-02&sr=b&
↪sig=<signature>&sp=rcw
```

If both the shared account key and the SAS token aren't provided, Azure will attempt to authenticate from a default identity chain. For more information, check the Azure documentation on the DefaultAzureCredential Class.

Creating a Google Cloud Storage bucket

For Google Cloud Storage, the `bucketUri` parameter uses the following format:

```
gs://<bucket-name>/<backup-name>
```

The credentials must be provided as files. If no authentication file is provided, the authentication will fall back to the application's default credential chain. To learn how to create credentials, check the Google Cloud Platform documentation on creating credentials for service accounts.

6.3.2 Creating a backup

As mentioned, backups can be either covering **all repositories** (full data backup) or only **selected existing repositories** (partial data backup), and they may also include the user accounts and settings.

- **Full data backups** will wait until all running updates are completed.
- **Partial data backups** will wait until running updates on the affected repositories are completed.

Note: Cluster backup creation is lock-free, meaning that by leveraging the multiple instances and quorum mechanism, the cluster can create a backup while simultaneously processing updates if the deployment has more than 2 nodes.

A GraphDB instance can be backed up using the `/rest/recovery/backup` endpoint. To create a backup, simply POST an HTTP request as shown further down below.

Tip: The backup is created in a `.tar` archive. If the backup creation has been successful, the archive should contain a `.success` file. If there is no such success file, the backup is corrupted and *GraphDB will not be able to restore data from it*.

See also:

- *Creating a cloud backup*

6.3.2.1 Backup options

The following parameters can be configured when creating a backup:

Option	Description
`repositories`	List of repositories to be backed up. Specified as JSON in the request body. • If the parameter is missing, all repositories will be included in the backup. • If it is an empty list (`[]`), no repositories will be included in the backup. • Otherwise, the repositories from the list will be included in the backup.
`backupSystemData`	Determines whether user account data such as user accounts, saved queries, or visual graphs, among other. should be included in the backup. Specified as JSON in the request body. Boolean, the default value is `false`.

6.3.2.2 Full data backup

Here is an example curl request for full data backup creation **without system data** (i.e., user accounts and settings):

```
curl -X POST -OJ -H 'Content-Type: application/json' '<base_url>/rest/recovery/backup'
```

This does the following:

- Backs up all data in all repositories.
- Does not include user accounts and settings because `backupSystemData = false` by default (see the above *Backup options*).
- Creates the backup as a new file of the type `backup-yyyy-mm-dd-hh-mm-ss.tar`.

Note: This is an archive file that you do not need to extract — it is to be used as is.

To set the name of the backup yourself, replace `-OJ` with `--output <backup-name>`, i.e.:

```
curl -X POST --output <backup-name> -H 'Content-Type: application/json' '<base_url>/rest/recovery/backup
↪'
```

6.3.2.3 Partial data backup

Here is an example curl request for partial data backup creation **without system data**:

```
curl -X POST -OJ -H 'Content-Type: application/json' -d '{
   "repositories":["<repo_name>"]
}' '<base_url>/rest/recovery/backup'
```

Which does the following:

- Backs up one or more repositories that are explicitly named.
- Does not include user accounts and settings as `backupSystemData = false` by default.

You can also use `--output <backup-name>` instead of `-OJ` if you want to customize the name of the backup as shown *above*.

Note: If a POST request does not include a list of repositories for backup, it will automatically create a full data backup.

6.3.2.4 Full data and system backup

Here is an example curl request for full data and system backup creation **with system data**:

```
curl -X POST -OJ -H 'Content-Type: application/json' -d '{
   "backupSystemData": true
}' '<base_url>/rest/recovery/backup'
```

Which does the following:

- Backs up all data in all repositories.
- Backup includes user accounts and settings as `backupSystemData = true` is explicitly provided.

6.3.2.5 Backup of only system data

Here is an example curl request for creating a backup of system data only:

```
curl -X POST -OJ -H 'Content-Type: application/json' -d '{
   "repositories" : [], "backupSystemData": true
}' '<base_url>/rest/recovery/backup'
```

Which does the following:

- Backup includes user accounts and settings as `backupSystemData = true` is explicitly provided.
- No repositories are included in the backup as `repositories` is an empty list (`repositories: []`).

> **Note:** If this parameter is not provided, all repositories will be included in the backup.

6.3.2.6 Creating a cloud backup

The GraphDB instance uses a different endpoint when creating a backup saved in the cloud — `/rest/recovery/cloud-backup`. This endpoint takes several parameters.

- `Repositories` (optional): List of repositories to be backed up. If nothing is passed for it, the default values of the options will be used. See the section on *backup options* above for further details.
- `backupSystemData` (optional): Determines whether user account data. If nothing is passed for it, the default values of the options will be used. See the section on *backup options* above for further details.
- `bucketUri` (required)
- `authenticationFile` (optional)

Below are example cURL request for full data backup creation with system data.

Creating a backup in Amazon S3 and compatible services

```
curl -X POST --header 'Content-Type: multipart/form-data' --header 'Accept: application/json' \
  -F 'params={"repositories" :["<repo-name>"], "backupSystemData" :<boolean>,
  "bucketUri": "s3:///<bucket_name>/<backup_name>?region=<region>&AWS_ACCESS_KEY_ID=<key_id>&AWS_SECRET_
↪ACCESS_KEY=<key>"}' \
  '<base_url>/rest/recovery/cloud-backup'
```

Creating a backup in Azure Blob

```
curl -X POST --header 'Content-Type: multipart/form-data' --header 'Accept: application/json' \
  -F 'params={"repositories": ["<repo-name>"],"backupSystemData": <boolean>,
  "bucketUri": "az://<container-name>/<backup-name>?blob_storage_account=<storage_account_name>&blob_
↪access_key=<key>"}' \
  '<base_url>/rest/recovery/cloud-backup'
```

Creating a backup in Google Cloud Platform

```
curl -X 'POST' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'params={"repositories":["<repo-name>"],"backupSystemData":<boolean>,"bucketUri":"gs://<bucket>/
↪<backup-name>"}' \
  -F 'authenticationFile=@/path/to/file/google-credentials.json' \
  '<base_url>/rest/recovery/cloud-backup'
```

Tip: The examples for *Full data backup*, *Partial data backup*, *Full data and system backup*, and *Backup of only system data* shown above are also valid for the cloud backup. As long as the cloud backup is provided with the same parameters and the `bucketUri` is valid, the resulting backup `.tar` file should be the same.

6.3.3 Restoring from a backup

A GraphDB instance or cluster can be restored to a backed-up state through the `/rest/recovery/restore` endpoint.

The recovery procedure **in the cluster** is treated as a simple update as it leverages the Raft protocol that allows a set of distributed nodes to act as one.

Warning: If the backup is corrupted, GraphDB will not be able to restore data from it. A healthy backup should contain a `.success` file in its package. If there is no such file, the backup is considered corrupted.

Note: It is recommended to perform cluster transaction log *truncate* operations **after a successful data restore**, as the transaction log will use more storage space upon a backup/restore procedure.

To restore a backup, simply POST an HTTP request as shown below.

See also:

- *Restoring a cloud backup*

6.3.3.1 Restore options

The following parameters can be configured when restoring from a backup:

Option	Description
`repositories`	List of repositories to recover from the backup. Specified as JSON in the request body. • If the parameter is missing, all repositories that are in the backup will be restored. • If it is an empty list (`[]`), no repositories from the backup will be restored. • Otherwise, the repositories from the list will be restored.
`restoreSystemData`	Determines whether GraphDB should restore user account data such as user accounts, saved queries, visual graphs etc. from a backup or continue with the their current state. Specified as JSON in the request body. If no system data is found in the backup, an error will be returned. Boolean, the default is `false`.
`removeStaleRepositories`	Cleans other existing repositories on the GraphDB instance where the restore is done. The default is `false`, meaning that no repositories will be cleaned.

6.3.3.2 Full data restore preserving other repositories

If we have successfully created a backup and want to completely revert to the backed-up state while preserving the existing repositories on the instance where we are restoring, we can use the below curl request example. No additional parameters are provided, meaning that defaults are applied.

```
curl -X POST '<base_url>/rest/recovery/restore' \
-H 'Content-Type: multipart/form-data' \
-F 'params={
        };type=application/json' \
-F file=@./<full-data-backup-name.tar>
```

Note: The `full-data-backup-name.tar` file must be a *Full data backup*.

6.3.3.3 Full data restore with replace

We can also apply a backup and remove repositories that are not restored from it.

```
curl -X POST '<base_url>/rest/recovery/restore' \
-H 'Content-Type: multipart/form-data' \
-F 'params={
        "removeStaleRepositories": true \
    };type=application/json' \
-F file=@./<full-data-backup-name.tar>
```

What this does:

- Removes other repositories on the instance where the backup is applied as `removeStaleRepositories = true`.
- Does a full data restore as the `repositories` parameter is not provided.

6.3.3.4 Partial data restore

Here, we need to provide the names of the repositories that we want to restore as values for the `repositories` parameter.

```
curl -X POST '<base_url>/rest/recovery/restore' \
-H 'Content-Type: multipart/form-data' \
-F 'params={
        "repositories" : ["<repo-name>"] \
    };type=application/json};type=application/json' \
-F file=@./<full-data-backup-name.tar>
```

6.3.3.5 Restoring only system data

To restore only the system data from a backup, we can use the following curl request:

```
curl -X POST '<base_url>/rest/recovery/restore' \
-H 'Content-Type: multipart/form-data' \
-F 'params={
        "repositories" : [], \
        "restoreSystemData": true \
    };type=application/json}' \
-F file=@./<full-data-system-backup-name.tar>
```

What this does:

- User account data is restored as `restoreSystemData = true`.
- No repositories are restored as the `repositories` parameter is an empty list (`[]`).

> **Note:** The `full-data-system-backup-name.tar` file must contain system data, i.e., the backup must be created with `backupSystemData = true` as shown in the section *Full data and system backup*.

6.3.3.6 Restoring from a cloud backup

The GraphDB instance uses a different endpoint when restoring from a backup saved on cloud storage — `/rest/recovery/cloud-restore`.

Below are example cURL request for applying a backup and removing all repositories that are not restored from it. In case you want to do a full backup (as opposed to a partial one), omit the `repositories` parameter.

Restoring a backup in Amazon S3 and compatible services

```
curl -X POST --header 'Content-Type: multipart/form-data' --header 'Accept: application/json' \
  -F 'params={"repositories":["<repo-name>"],"removeStaleRepositories":<boolean>,"restoreSystemData":
↪<boolean>,
  "bucketUri":"s3:///<bucket_name>/<backup_name>?region=<region>&AWS_ACCESS_KEY_ID=<key_id>&AWS_SECRET_
↪ACCESS_KEY=<key>"}' \
  '<base_url>/rest/recovery/cloud-restore'
```

Restoring a backup in Azure Blob

```
curl -X POST --header 'Content-Type: multipart/form-data' --header 'Accept: application/json' \
  -F 'params={"repositories":["<repo-name>"],"removeStaleRepositories":<boolean>,"restoreSystemData":
↪<boolean>,
  "bucketUri":"az://<container-name>/<backup-name>?blob_storage_account=<storage_account_name>&blob_
↪access_key=<key>"}' \
  '<base_url>/rest/recovery/cloud-restore'
```

Restoring a backup in Google Cloud Storage

```
curl -X 'POST' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'params={"repositories":["<repo-name>"],"removeStaleRepositories":<boolean>,"restoreSystemData":
↪<boolean>,"bucketUri":"gs://<bucket>/<backup-name>"}' \
  -F 'authenticationFile=@/path/to/file/google-credentials.json' \
  '<base_url>/rest/recovery/cloud-restore'
```

> **Tip:** The examples for *Full data restore preserving other repositories*, *Full data restore with replace*, *Partial data restore*, and *Restoring only system data* shown above are also valid for the cloud restore. As long as the cloud restore is provided with the parameters and the `bucketUri` is a valid GraphDB backup file, the resulting restore should be the same.

6.3.4 Backup compatibility between different versions of GraphDB

When restoring a backup, compatibility between different versions of GraphDB mostly depends on whether you are restoring between different *patch versions, minor versions, or major versions of GraphDB*. GraphDB strives to present backward backup compatibility in most cases and generally you should be able to restore backups made with older versions of GraphDB than the one you are currently using, but this may not always be the case.

6.3.4.1 Backup-restore compatibility between different patch versions

We always allow unconditional backward and forward backup-restore between patch versions of the same minor release. For example, backup from any GraphDB 11.3 patch release, such as 11.3.3, should be restored on any 11.3.x version, even if it is a previous patch version.

6.3.4.2 Backup-restore compatibility between different minor versions

Forward backup-restore is supported between minor versions. A backup created on 11.2.x should be restored on any patch release for 11.3, 11.4, and so on.

Backward backup-restore between minor versions is supported by default, but is not unconditional. For some minor versions, support for backward backup-restore may be dropped to accomodate for fixing major and critical bugs, or for improvements and new feautres. In such case this will be explicitly documented in the *Migrating GraphDB configurations* page.

6.3.4.3 Backup-restore compatibility between different major versions

Forward backup-restore is supported by default between a new major version and the latest previous minor release by default. Exceptions may occur and will be explicitly documented in the *Migrating GraphDB configurations* page. For example you can restore from 10.8.x to any patch release for GraphDB 11 (10.8 is the last 10 minor version).

Backup-restore skipping a major version is not supported. In other words, restoring a backup from GraphDB 9 to GraphDB 11 is not possible.

Backward backup-restore is not supported between major versions. This will **not** be documented explicitly for each new major version.

6.3.5 Monitoring your recovery operations

You can monitor your backups through *Monitor* ‣ *Backup and Restore*. The backup monitoring interface displays the backups and restores that are currently underway, and displays additional information, such as the recovery operation type (backup or restore), the user who initiated the procedure, the affected repositories, how much time has elapsed since the procedure was initiated, and the snapshot options.

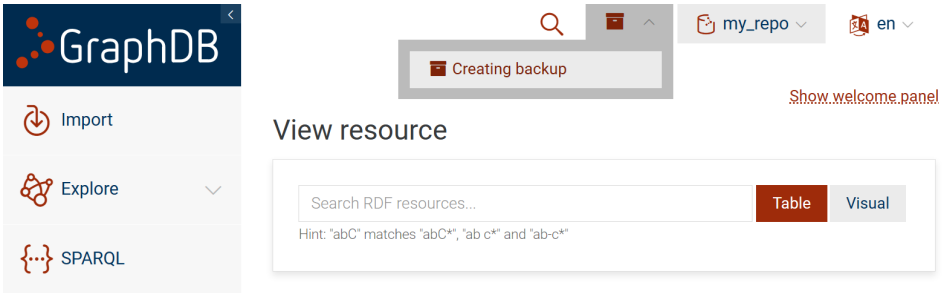
This interface also allows you to temporarily *Pause* updates of the table so that you can copy text from it.

Pause

Id	Username	Recovery operation type	Affected repositories	Lifetime	Snapshot options
7	admin	Creating backup	acl_rules_example test-repo	3s	cleanDataDir : false withRepositoryData : true withSystemData : false

> **Warning:** The *Pause* button doesn't pause the recovery operation itself — it merely freezes the information presented in Workbench and prevents the table from updating.

You can also access the *Backup and Restore* interface through the *The Workbench notification area* notification area on the top of every Workbench view.



6.3.5.1 Monitoring your recovery operations through cURL

Track backup operations

```
GET /rest/monitor/backup
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/monitor/backup'
```

This command will return the current recovery operation. The response will specify whether the operation is a backup or a restore.

6.4 Monitoring and troubleshooting

6.4.1 Request tracking

Tracking a single request through a distributed system is an issue due to the scattered nature of the logs. Therefore, GraphDB offers the capability for tracking particular request ID headers, or generates those itself if need be. This allows for easier auditing and system monitoring. Headers will be intercepted when a request comes into the database and passed onwards together with the response. Request tracking is turned off by default, and can be enabled by adding `graphdb.append.request.id.headers=true` to their `graphdb.properties` file. The value is already present in the default configuration file, but needs to be uncommented to work.

By default, GraphDB scans all incoming requests for an `X-Request-ID` header. If no such header exists, it assigns to the incoming request a random ID in the UUID type 5 format.

Some clients and systems assign alternative names to their request identifiers. Those can be listed in the following format:

```
graphdb.request.id.alternatives=my-request-header-1, outside-app-request-header
```

6.4.2 Monitoring your AWS deployment

GraphDB exposes several endpoints that return metrics in the Prometheus text format. These endpoints could be scraped using the CloudWatch agent so that you can monitor the metrics in CloudWatch and create alarms based on them.

In order for the CloudWatch agent to have permission to push metrics to CloudWatch, your EC2 instances will need to have the `CloudWatchAgentServerPolicy`, `CloudWatchAdminPolicy` and `CloudWatchFullAccessV2` permissions attached to their instance profile. See the official AWS documentation as well as the GraphDB documentation on *Creating an IAM role and attaching policies* for more information.

Once the required permissions are attached, follow these steps to push metrics to CloudWatch:

1. Install CloudWatch agent on the EC2 instances
2. Create a `/etc/prometheus/prometheus.yml` Prometheus config on each instance and paste the following script:

```
global:
scrape_interval: 1m
scrape_timeout: 10s
scrape_configs:
- job_name: graphdb_infrastructure_monitor
    metrics_path: /rest/monitor/infrastructure
    scrape_interval: 5s
    static_configs:
    - targets: [ 'localhost:7200' ]
- job_name: graphdb_structures_monitor
    metrics_path: /rest/monitor/structures
    scrape_interval: 5s
    static_configs:
    - targets: [ 'localhost:7200' ]
```

Note: This step is necessary so that the CloudWatch agent knows which endpoints to scrape.

Tip: Check the *Monitoring and troubleshooting* documentation for more information on the various metrics and endpoints.

3. Create a file named `config.json`. You can use the example below:

```
{
    "agent": {
        "debug": true
    },
    "logs":{
    "metrics_collected":{
        "prometheus":{
            "log_group_name": "graphdb",
            "prometheus_config_path": "/etc/prometheus/prometheus.yml",
            "emf_processor": {
                "metric_declaration_dedup": true,
                "metric_namespace": "CWAgent-Prometheus",
                "metric_unit":{
                "graphdb_class_count": "Count"
                },
                "metric_declaration":[
                    {
                        "source_labels": [
                            "job"
```

(continues on next page)

(continued from previous page)

```
                        ],
                        "label_matcher": "graphdb_hw_monitor",
                        "dimensions": [["host"]],
                        "metric_selectors":[
                            "^graphdb_class_count$"
                        ]
                }
                ]
            }
        }
    }
    }
}
```

For a complete example, check our Terraform module.

Note: The agent also requires a configuration file for itself. This is where we reference the prometheus configuration and specify the metrics that we are interested in. Check the AWS documentation for more details.

4. Use the following command to start the CloudWatch agent:

```
amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -s -c file:config.json
```

5. Go to the CloudWatch console and check if the metrics are being sent over

6.4.3 Monitoring your Azure deployment

GraphDB monitoring in Azure is done through the Azure Application Insights resource to achieve near real time metrics and alerts.

Note: We strongly recommend using the Terraform module when *deploying GraphDB in Azure* as it also automates the process of setting up the monitoring of your GraphDB Azure deployment. You can find the module in our GitHub repository. The Terraform module is also available in the Terraform registry.

6.4.3.1 Configuring the monitoring for GraphDB in Azure

Prerequisites

- Azure Application Insights
- `applicationinsights.json` configuration file located within the `/opt/graphdb/` directory of each of your instances
- Log Analytics workspace (created and configured automatically while deploying GraphDB)

Note: If our VM image is not used, you will also need to setup the App insights agent and configure the `GDB_JAVA_OPTS` environment variable. See the GraphDB installation script and an example environment variable configuration in our GitHub repository.

Deploying Azure Application Insights

1. Navigate to the *Resource Group* which contains the resources for your GraphDB deployment.
2. Click on + *Create* and search for *Application Insights*. Select it, then click on *Create*.
3. Provide a unique name, and deploy the resource in the same region as your GraphDB deployment.

Note: The *Application Insights* resource will automatically create its own Log Analytics Workspace.

Once the Application Insights resource has been deployed, you will need to obtain the Connection string from it.

1. Navigate to the newly created Application Insights resource, and go to *Overview* ▸ *Essentials* ▸ *Connection string*.
2. Copy the connection strings. You will need it while creating the configuration file in the next step.

Creating the config file

In order for the Application Insights resource to be able to access and read the metrics provided by GraphDB, you need to create a JSON configuration file inside each one of your instances.

1. For each instance, create a JSON file named `applicationinsights.json` inside the `/opt/graphdb/` directory.
2. Provide the connection string from the *Application Insights* resource.
3. Provide the name of the current instance.
4. Set the `sampling`, `instrumentation` and other configuration properties.
5. Once you have created a configuration file in each of your instances, restart your GraphDB deployment. You will now be able to use Application Insights to monitor your GraphDB instances.

Warning: Since the volume is not persisted, the JSON configurations file must be provided upon instance restart or recreation. This can be automated and handled by a user data script.

You can use the example configuration file below as a reference when creating your `applicationinsights.json` files.

```
{
    "role": {
        "name": "<CURRENT_INSTANCE_NAME>"
      },
    "connectionString": "<APPLICATION_INSIGHTS_CONNECTION_STRING>",
    "sampling": {
      "percentage": 100
  },
    "instrumentation": {
        "logging": {
            "level": "WARN"
        }
    },
    "preview": {
        "captureLogbackMarker": true,
        "captureLog4jMarker": true,
        "captureLogbackCodeAttributes": true,
        "sampling": {
            "overrides": [
              {
```

(continues on next page)

(continued from previous page)

```
                "telemetryType": "request",
                "percentage": 0
                "attributes": [
                   {
                     "key": "rpc.service",
                     "value": ".*cluster.*",
                     "matchType":  "regexp"
                   }
                ]
              }
            ]
        }
    }
}
```

Note: If you set sampling to below 100, we recommend you log full data from specific endpoints such as the `rest/recovery` endpoint.

To stop collecting data from the cluster gRPC communication, you can add the following in a sampling overrides block:

```
{
  "telemetryType": "request",
  "percentage": 0
  "attributes": [
      {
        "key": "rpc.service",
        "value": ".*cluster.*",
        "matchType":  "regexp"
      }
  ]
}
```

To stop collecting data about dependencies, add:

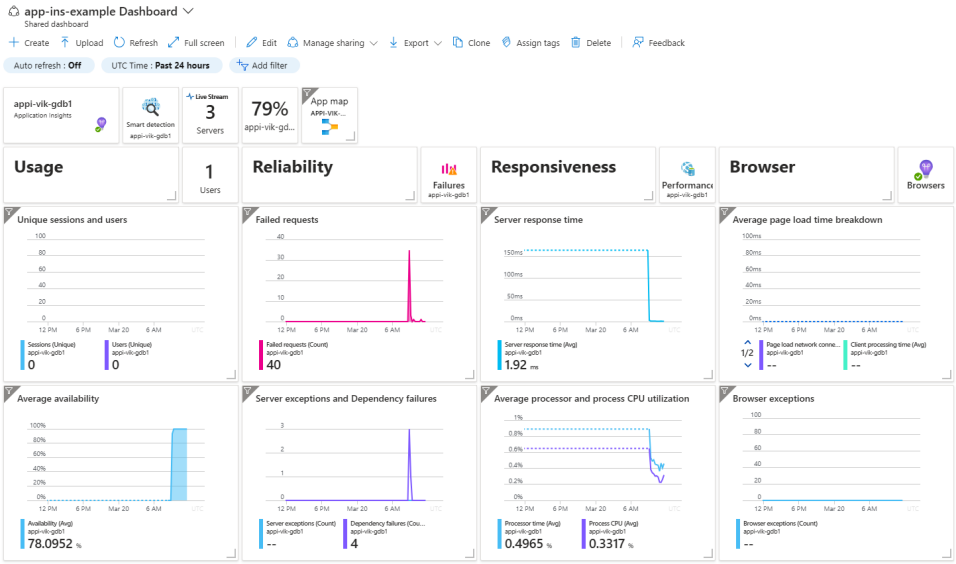
```
{
  "telemetryType": "dependency",
  "percentage": 0
}
```

6.4.3.2 Using Azure Application Insights to monitor your GraphDB deployment

Once you have configured your monitoring environment, you can now use Application Insights to access various metrics and data.

Note: In case you have used the Terraform module to set up your GraphDB Azure deployment, please check the applicationinsights.json file for more information on the default configuration.

One of the main monitoring tools of Application Insights is the Application Dashboard, which displays the most important information about your deployment. You can access the Dashboard by navigating to *Application Insights* -‣ *Overview*, and clicking on the *Application Dashboard* button.



The following table lists several other noteworthy features which allow you to monitor various points of interest about the deployment usage, performance, availability and failures.

Feature	Description
Live metrics	Provides live metrics from the GraphDB Java process, as well as incoming request telemetry. This is particularly helpful when monitoring a cluster, as it enables the simultaneous monitoring of all nodes, eliminating the necessity to individually log in to each separate node.
Transaction search	Allows the search of specific transaction or log message from GraphDB.
Availability tests	Enables the monitoring of how the availability tests respond in a given time frame. Various properties can be configured, such as the coverage of the SLA report, the length of the time frame, and the failure threshold. **Note:** • The availability tests rely on the public address and check the liveliness from 5 different regions. • By default the availability target is set to 99.5%.
Failures	Displays failed operations, as well as a summary of the status codes returned. **Tip:** You can drill down. However, the drill down might not be avialable if the requests are filtered in the applicationinsights.json file. For example, failed requests for static content will not be displayed in the Overall view but will still be included in the total count.
Performance	Displays the performance of the monitored endpoints, distribution of requests over time, and a summary for common properties those requests share.
Smart detection	Displays all issues detected by the Azure smart detection. **Tip:** You can see the default settings the *Terraform script configures in the table below*.
Alerts	Displays all fired alerts. The alerts can be configured to self-resolve if the set condition is no longer met. **Tip:** The Terraform module also configures several alerts: • Low disk space warning (Log search alert rule) • Replication warning (Log search alert rule) • Availability alert (Metric alert rule) • Memory alert (Metric alert rule) If you are not using the Terraform module, you may want to consider creating these resources on your own.
Workbooks	Application insights has predefined workbooks which cover detailed analytics for Availability, Failures, Performance and more. **Tip:** You can edit the provided workbooks.

Default settings for Smart detection

By default, all Smart detection rules are enabled. When using the Terraform script to deploy GraphDB in Azure, it disables some of the rules irrelevant to running GraphDB, as outlined below:

Enabled	Name	Severity
×	Slow page load time	Information
✓	Slow server response time	Information
×	Long dependency duration	Information
✓	Degradation in server response time	Information
×	Degradation in dependency duration	Information
×	Degradation in trace severity ratio (preview)	Information
✓	Abnormal rise in exception volume (preview)	Information
✓	Potential memory leak detected (preview)	Information
✓	Potential security issue detected (preview)	Information
✓	Abnormal rise in daily data volume (preview)	Information

6.4.4 Repository health checks

The GraphDB health check endpoint is at `http://localhost:7200/repositories/<repo_name>/health`.

Possible responses:

- HTTP status `200`: the repository is healthy
- HTTP status `206`: the repository needs attention but it is not something critical
- HTTP status `500`: the repository is inconsistent, i.e., some checks failed

6.4.4.1 Possible values for health checks and their meaning

Value	Description
`repository-state`	Checks the state of the repository. Returns message RUNNING, STARTING, or INACTIVE. RUNNING and INACTIVE result in `green` health, and all other states result in `yellow` health.
`read-availability`	Checks whether the repository is readable.
`write-availability`	A `green` status means that you are either not in a cluster or that all nodes in the cluster are available. A `yellow` status shows that there are some unavailable nodes in the cluster, but a majority quorum of available nodes is present so that you can execute successful updates. A `red` status shows that a majority of the nodes are not available, so you cannot execute any updates in your repository.
`storage-folder`	Checks if there is at least 5 percent writable space left for the storage folder. This percentage can be controlled with the system parameter `graphdb.health.minimal.free.storage.fatal`.
`long-running-queries`	Checks if there are queries running longer than 20 seconds. The time can be controlled with the system parameter `health.max.query.time.seconds`. If a query is running for more than 20 seconds, it is either a slow one, or there is a problem with the database.
`predicates-statistics`	Checks if the predicate statistics contain correct values.
`plugins`	Provides aggregated health checks for the individual plugins.

6.4.4.2 Aggregated health checks

The aggregated GraphDB health checks include checks for dependent services and components as plugins and connectors.

Each connector plugin is reported independently as part of the composite "plugins" check in the repository health check. Each connector's check is also a composite where each component is an individual connector instance.

The output may look like this:

```
{
  "name": "wine",
  "status": "green",
  "components": [
    {
      "name": "read-availability",
      "status": "green"
    },
    {
      "name": "storage-folder",
      "status": "green"
    },
    {
      "name": "long-running-queries",
      "status": "green"
    },
    {
      "name": "predicates-statistics",
      "status": "green"
    },
    {
      "name": "write-availability",
      "status": "green",
      "message": "All nodes are in sync"
    },
    {
      "name": "plugins",
      "status": "yellow",
      "components": [
        {
          "name": "elasticsearch-connector",
          "status": "green",
          "components": []
        },
        {
          "name": "lucene-connector",
          "status": "yellow",
          "components": [
            {
              "name": "my_index",
              "status": "green",
              "message": "query took 0 ms, 5 hits"
            },
            {
              "name": "my_index2",
              "status": "yellow",
              "message": "query took 0 ms, 0 hits"
            }
          ]
        },
        {
          "name": "solr-connector",
          "status": "yellow",
```

(continues on next page)

(continued from previous page)

```
          "components": [
            {
              "name": "my_index",
              "status": "green",
              "message": "query took 7 ms, 5 hits"
            },
            {
              "name": "my_index2",
              "status": "yellow",
              "message": "query took 5 ms, 0 hits"
            }
          ]
        }
      ]
    }
  ]
}
```

An individual check run involves sending a query for all documents to the connector instance, and the result is:

- `green` — more than zero hits
- `yellow` — zero hits or failing shards (shards check only for Elasticsearch and OpenSearch)
- `red` — unable to execute query

In all of these cases, including the `green` status, there is also a message providing details, e.g., "query took 15 ms, 5 hits, 0 failing shards".

Running health checks

To run the health checks for a particular repository, in the example `myrepo`, execute the following cURL command:

```
curl 'http://localhost:7200/repositories/myrepo/health'
```

6.4.4.3 Passive health checks

In passive check mode, the repository state will be compared to determine if it is safe to do an active check.

- Immediate passive: Activated by passing `?passive` to the health endpoint.
  - If the state is RUNNING, do an active check.
  - If the state is something else (e.g., INACTIVE or STARTING), return immediately with a simple check that only lists the state.
- Delayed passive (if needed): Tries to get the repository for up to N seconds. Activated by passing `?passive=N` to the endpoint, where N is a timeout in seconds.
  - If successful: Runs an active check.
  - If timeout: Return with a simple check that only lists the state.

Example health from a passive check (repo has never requested since GraphDB restart):

```
{
   "status" : "green",
   "components" : [
      {
         "status" : "green",
         "name" : "repository-state",
         "message" : "INACTIVE"
```

(continues on next page)

(continued from previous page)

```
      }
   ],
   "name" : "test"
}
```

Example health from a passive check (repo is currently initializing):

```
{
   "status" : "yellow",
   "components" : [
      {
         "status" : "yellow",
         "name" : "repository-state",
         "message" : "STARTING"
      }
   ],
   "name" : "test"
}
```

6.4.5 Low disk space health checks

The GraphDB engine implements disk space validation health checks in order to avoid failure scenarios and unexpected behavior due to low disk space. These checks preserve GraphDB's integrity even when a machine runs out of disk space.

The disk space health checks mechanism is enabled by default and is controlled by the `graphdb.health.minimal.free.storage.enabled` and `graphdb.health.minimal.free.storage.asyncCheck` properties. If you want to disable the mechanism, both properties need to be disabled.

Whenever a certain threshold of available disk space is reached, GraphDB logs a caution specifying the free space availability. There are two levels of warnings logged, followed with a **fatal** state which halts all transactions and requires manual addressing.

1. **Warnings** are issued whenever the available disk space falls below 20% or less.
2. **Errors** are issued whenever the available disk space falls below 10% or less.
3. **Fatal** state is reached whenever the available disk space fall below 5% or 1GB, whichever comes first. At this point GraphDB will prevent new transactions from being started and log a **fatal** state.

The default values of the three admonition levels can be changed through their *respective configuration properties*.

Note: Only the percentage value of the **critically low** threshold can be changed. The 1GB threshold cannot be changed.

6.4.5.1 Recovering from a fatal state

Whenever the available disk space falls below the critically low threshold, GraphDB prevents new transactions from being started until either enough disk space has been freed, or enough additional disk space is allocated to the drive (if running on a dynamic drive).

Tip: To free additional disk space, you can delete server logs and large server files.

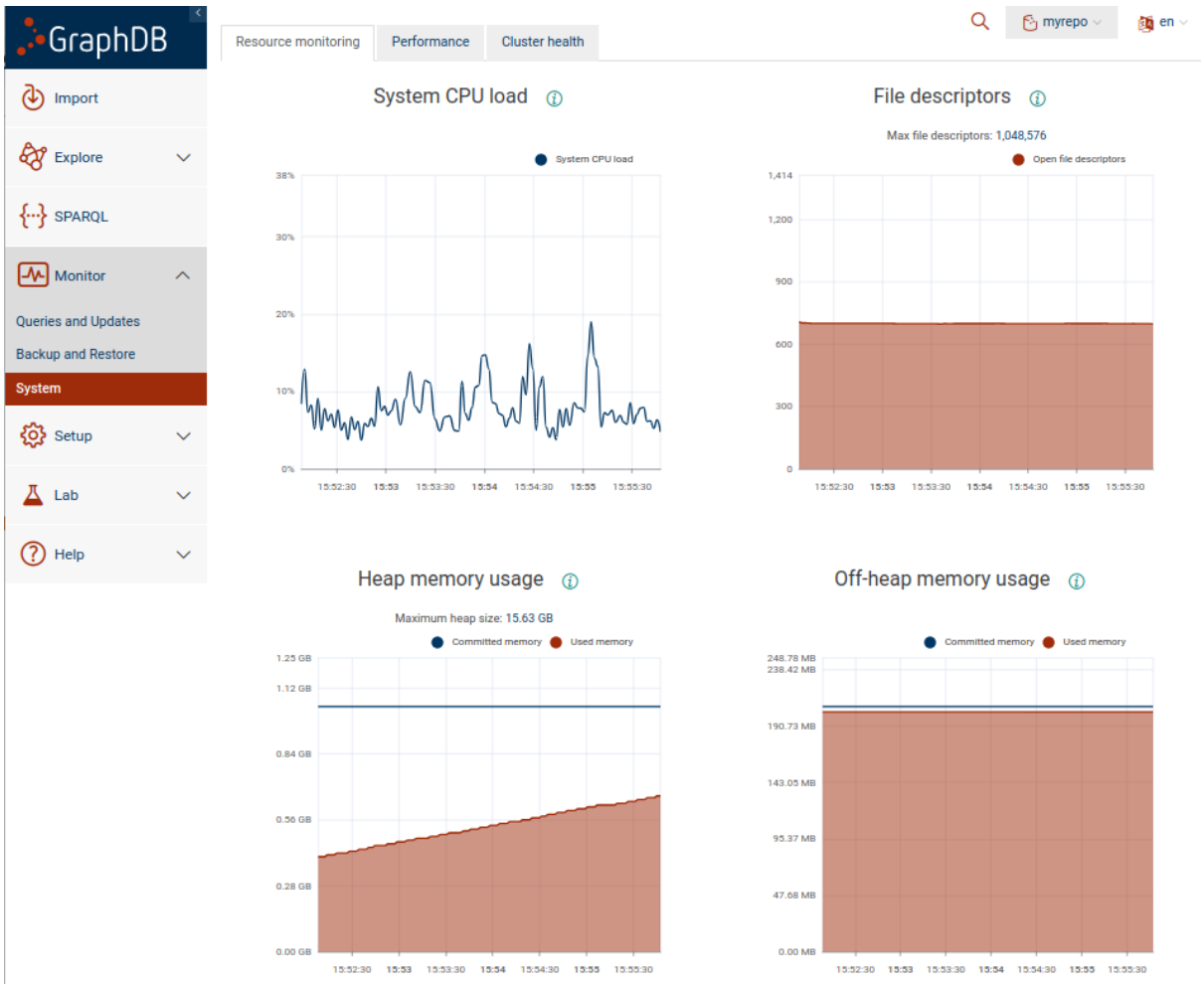
6.4.6 System monitoring

GraphDB offers several options for system monitoring described in detail below.

6.4.6.1 Workbench monitoring

You can monitor the most important hardware information as well as other application-related metrics from the following three tabs under *Monitor* ‣ *System* in the GraphDB Workbench.

- *Resource monitoring*: system CPU load, file descriptors, heap memory usage, off-heap memory usage, and disk storage.
- *Performance* (per repository): queries, global page cache, entity pool, and transactions and connections.
- *Cluster health* (in case a cluster exists).



6.4.6.2 Prometheus monitoring

The GraphDB REST API exposes several *monitoring endpoints* suitable for scraping by Prometheus. They return a suitable data format when the request has an `Accept` header of the type `text/plain`, which is the default type for Prometheus scrapers.

GraphDB structures monitoring API

The `/rest/monitor/structures` endpoint enables you to monitor GraphDB structures — the *global page cache* and the *entity pool*. This provides a better understanding of whether the current GraphDB configuration is optimal for your specific use case (e.g., repository size, query complexity, etc.)

The current state of the global page cache and the entity pool are returned via the following metrics:

Parameter	Description
`graphdb_cache_hit`	GraphDB's global page cache hits count. Along with the global page cache miss count, this metric can be used to diagnose a small or oversized global page cache. • In ideal conditions, the percentage of hits should be over 96%. • If it is below 96%, it might be a good idea to increase its size. • If it is over 99%, it might be worth experimenting with a smaller global page cache size.
`graphdb_cache_miss`	GraphDB's global page cache miss count.

Infrastructure statistics monitoring API

The `/rest/monitor/infrastructure` endpoint enables you to monitor GraphDB's infrastructure so as to have better visibility of the hardware resources usage. It returns the most important hardware information and several application-related metrics:

Parameter	Description
`graphdb_open_file_descriptors`	Count of currently open file descriptors. This helps diagnose slow-downs of the system or a slow storage if the number remains high for a longer period of time.
`graphdb_cpu_load`	Shows the current CPU load for the entire system in %.
`graphdb_heap_max_mem`	Maximum available memory for the GraphDB instance. Returns `-1` if the maximum memory size is undefined.
`graphdb_heap_init_mem`	Initial amount of memory (controlled by `-Xms`) in bytes.
`graphdb_heap_committed_mem`	Current committed memory in bytes.
`graphdb_heap_used_mem`	Current used memory in bytes. Along with the rest of the memory-related properties, this can be used to detect memory issues.
`graphdb_mem_garbage_collections_count`	Count of full garbage collections from the starting of the GraphDB instance. This metric is useful for detecting memory usage issues and system "freezes".
`graphdb_nonheap_init_mem`	Off-heap initial memory in bytes.
`graphdb_nonheap_max_mem`	Maximum direct memory. Returns `-1` if undefined.
`graphdb_nonheap_committed_mem`	Current off-heap committed memory in bytes.
`graphdb_nonheap_used_mem`	Current off-heap used memory in bytes.
`graphdb_data_dir_used`	Used storage space on the partition where the data directory sits, in bytes. This is useful for detecting a soon-out-of-hard-disk-space issue along with the free storage metric.
`graphdb_data_dir_free`	Free storage space on the partition where the data directory sits, in bytes.
`graphdb_logs_dir_used`	Used storage space on the partition where the logs directory sits, in bytes. This is useful for detecting a soon-out-of-hard-disk-space issue along with the free storage metric.
`graphdb_logs_dir_free`	Free storage space on the partition where the logs directory sits, in bytes.
`graphdb_work_dir_used`	Used storage space on the partition where the work directory sits, in bytes. This is useful for detecting a soon-out-of-hard-disk-space issue along with the free storage metric.
`graphdb_work_dir_free`	Free storage space on the partition where the work directory sits, in bytes.
`graphdb_threads_count`	Current used threads count.

Cluster statistics monitoring API

Via the `/rest/monitor/cluster` endpoint, you can monitor GraphDB's cluster statistics in order to diagnose problems and cluster slow-downs more easily. The endpoint returns several cluster-related metrics, and will not return anything if a cluster is not created.

Parameter	Description
`graphdb_leader_elections_count`	Count of leader elections from cluster creation. If there are a lot of leader elections, this might mean an unstable cluster setup with nodes that are not always properly operating.
`graphdb_failure_recoveries_count`	Count of total failure recoveries in the cluster from cluster creation. Includes failed and successful recoveries. If there are a lot of recoveries, this indicates issues with the cluster stability.
`graphdb_failed_transactions_count`	Count of failed transactions in the cluster.
`graphdb_nodes_in_cluster`	Total nodes count in the cluster.
`graphdb_nodes_in_sync`	Count of nodes that are currently in-sync. If a lower number than the total nodes count is reported, this means that there are nodes that are either out-of-sync, disconnected, or syncing.
`graphdb_nodes_out_of_sync`	Count of nodes that are out-of-sync. If there are such nodes for a longer period of time, this might indicate a failure in one or more nodes.
`graphdb_nodes_disconnected`	Count of nodes that are disconnected. If there are such nodes for a longer period of time, this might indicate a failure in one or more nodes.
`graphdb_nodes_syncing`	Count of nodes that are currently syncing. If there are such nodes for a longer period of time, this might indicate a failure in one or more nodes.

Query statistics monitoring API

Via the `/rest/monitor/repository/{repositoryID}` endpoint, you can monitor GraphDB's query and transaction statistics in order to obtain a better understanding of the slow queries, suboptimal queries, active transactions, and open connections. This information helps in identifying possible issues more easily.

The endpoint exists for each repository, and a scrape configuration must be created for each repository that you want to monitor. Normally, repositories are not created or deleted frequently, so the Prometheus scrape configurations should not be changed often either.

Note: In order for GraphDB to be able to return these metrics, the repository must be initialized.

The following metrics are exposed:

Parameter	Description
`graphdb_slow_queries_count`	Count of slow queries executed on the repository. The counter is reset when a GraphDB instance is restarted. If the count of slow queries is high, this might indicate a setup issue, unoptimized queries, or not good enough hardware.
`graphdb_suboptimal_queries_count`	Count of queries that the GraphDB engine was not able to evaluate and were sent for evaluation to the RDF4J engine. A too high number might indicate that the queries typically used on the repository are not optimal.
`graphdb_active_transactions`	Count of currently active transactions.
`graphdb_open_connections`	Count of currently open connections. If this number stays high for a longer period of time, it might indicate an issue with connections not being closed once their job is done.
`graphdb_entity_pool_reads`	GraphDB's entity pool reads count. Along with the entity pool writes count, this metric can be used to diagnose a small or oversized entity pool.
`graphdb_entity_pool_writes`	GraphDB's entity pool writes count.
`graphdb_epool_size`	Current entity pool size, i.e., entity count in the entity pool.

Prometheus setup

To scrape the mentioned endpoints in Prometheus, we need to add scraper configurations. Below is an example configuration for three of the endpoints, assuming we have a repository called "wines".

```
- job_name: graphdb_queries_monitor
metrics_path: /rest/monitor/repository/wines
scrape_interval: 5s
static_configs:
        - targets: [ 'my-graphdb-hostname:7200' ]
- job_name: graphdb_hw_monitor
metrics_path: /rest/monitor/infrastructure
scrape_interval: 5s
static_configs:
        - targets: [  'my-graphdb-hostname:7200' ]
- job_name: graphdb_structures_monitor
metrics_path: /rest/monitor/structures
scrape_interval: 5s
static_configs:
        - targets: [  'my-graphdb-hostname:7200' ]
```

Cluster monitoring

When configuring Prometheus to monitor a GraphDB cluster, the setup is similar with a few differences.

In order to get the information for each cluster node, each node's address must be included in the targets list.

The other difference is that another scraper must be configured to monitor the cluster status. This scraper can be configured in several ways:

- Scrape **only the external proxy** (which will always point to the current cluster leader) if it exists in the current cluster configuration.

  The downside of this method is that if for some reason, there is a connectivity problem between the external proxy and the nodes, it will not report any metrics.

- Scrape **the external proxy and all cluster nodes**.

  This method will enable you to receive metrics from all cluster nodes including the external proxy. This way, you can see the cluster status even if the external proxy has issues connecting to the nodes. The downside is that most of the time, each cluster will be duplicated for each cluster node.

- Scrape **all cluster nodes** (if there is no external proxy).

  If there is no external proxy in the cluster setup, the only option is to monitor all nodes in order to determine the status of the entire cluster. If you choose only one node and it is down for some reason, you would not receive any cluster-related metrics.

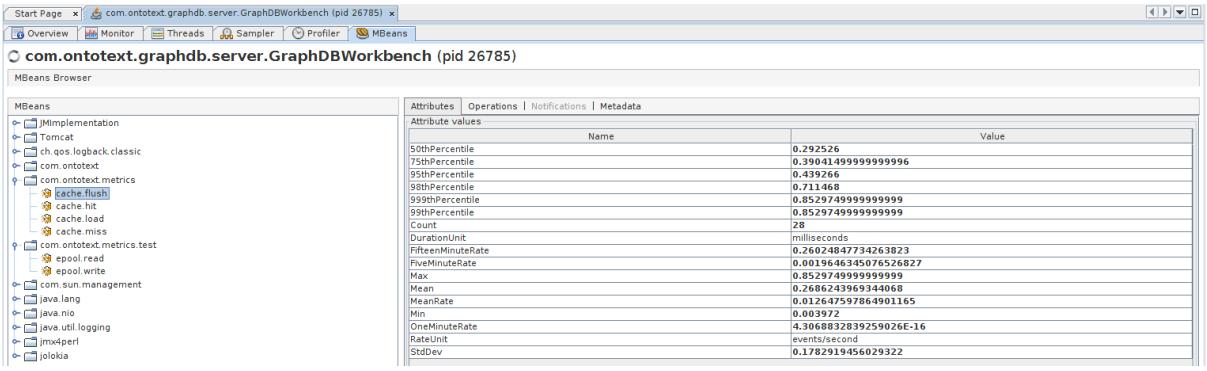
The scraper configuration is similar to the previous ones, with the only difference that the targets array might contain one or more cluster nodes (and/or external proxies). For example, if you have a cluster with two external proxies and five cluster nodes, the scraper might be configured to scrape only the two proxies like so:

```
- job_name: graphdb_cluster_monitor
metrics_path: /rest/monitor/cluster
scrape_interval: 5s
static_configs:
        - targets: [  'graphdb-proxy-0:7200',  'graphdb-proxy-1:7200' ]
```

As mentioned, you can also include some or all of the cluster nodes if you want.

6.4.6.3 JMX console monitoring

The database employs a number of metrics that help tune the memory parameters and performance. They can be found in the JMX console under the `com.ontotext.metrics` package. The global metrics that are shared between the repositories are under the top level package, and those specific to repositories — under `com.ontotext.metrics.<repository-id>`.



Page cache metrics

The *global page cache* provides metrics that help tune the amount of memory given for the page cache. It contains the following elements:

Parameter	Description
`cache.flush`	Counter for the pages that are evicted out of the page and the amount of time it takes for them to be flushed on the disc.
`cache.hit`	Number of hits in the cache. This can be viewed as the number of pages that do not need to be read from the disc but can be taken from the cache.
`cache.load`	Counter for the pages that have to be read from the disc. The smaller the number of pages is, the better.
`cache.miss`	Number of cache misses. The smaller this number is, the better. If you see that the number of hits is smaller than the misses, then it is probably a good idea to increase the page cache memory.

Entity pool metrics

You can monitor the number of reads and writes in the entity pool of each repository with the following parameters:

Parameter	Description
`epool.read`	Counter for the number of reads in the entity pool.
`epool.write`	Counter for the number of writes in the entity pool.

6.4.7 Diagnosing and reporting critical errors

It is essential to gather as many details as possible about an issue once it appears. For this purpose, we provide utilities that generate such issue reports by collecting data from various log files, JVM, etc. Using these issue reports helps us to investigate the problem and provide an appropriate solution as quickly as possible.

6.4.7.1 Running a system report

GraphDB provides an easy way to gather all important system information and package it as an archive that can be sent to graphdb-support@ontotext.com. Run the report using the GraphDB Workbench, or from the `generate-report` script in the `bin` directory of your distribution. The report is saved in the *GraphDB-Work*/report directory. There is always one report — the one that has been generated most recently.

Report content

- GraphDB version
- recursive directory list of the files in *GraphDB-Home* as `home.txt`
- recursive directory list of the files in *GraphDB-Work* as `work.txt`
- recursive directory list of the files in *GraphDB-Data* `data.txt`
- the 30 most recent logs files from *GraphDB-Logs* ordered by time of creation
- full copy of the content of *GraphDB-Conf*
- the output from jcmd `GC.class_histogram` as `jcmd_histogram.txt`
- the output from jcmd `Thread.print` as `thread_dump.txt`
- the System Properties for the GraphDB instance

- the repository configurations info as `system.ttl`. All repositories can be found in this file.
- the `owlim.properties` file for each repository if found. It exists only when the repository has been initialized at least once.

```
Archive:  graphdb-server-report-2022-06-06_092035+0300.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
       13  06-06-2022 09:20   gdb_version.txt
     9206  06-06-2022 09:20   home.txt
      498  06-06-2022 09:20   work.txt
     5788  06-06-2022 09:20   data.txt
      251  06-06-2022 09:20   logs/main-2022-06-06.log
       88  06-06-2022 09:20   logs/main-2022-06-04.log
        0  06-06-2022 09:20   logs/slow-query-log-2022-06-03.log
        0  06-06-2022 09:20   logs/query-log-2022-06-03.log
    42434  06-06-2022 09:20   logs/main-2022-06-03.log
        0  06-06-2022 09:20   logs/audit-log-2022-06-03.log
   211901  06-06-2022 09:20   logs/main-2022-05-31.log
        0  06-06-2022 09:20   logs/query-log-2022-05-30.log
        0  06-06-2022 09:20   logs/slow-query-log-2022-05-30.log
        0  06-06-2022 09:20   logs/audit-log-2022-05-30.log
   127028  06-06-2022 09:20   logs/main-2022-05-30.log
       88  06-06-2022 09:20   logs/main-2022-05-24.log
        0  06-06-2022 09:20   logs/query-log-2022-05-23.log
        0  06-06-2022 09:20   logs/slow-query-log-2022-05-23.log
        0  06-06-2022 09:20   logs/audit-log-2022-05-23.log
     1291  06-06-2022 09:20   logs/main-2022-05-23.log
        0  06-06-2022 09:20   logs/query-log-2022-05-19.log
        0  06-06-2022 09:20   logs/slow-query-log-2022-05-19.log
        0  06-06-2022 09:20   logs/audit-log-2022-05-19.log
     1370  06-06-2022 09:20   logs/main-2022-05-19.log
     8245  06-06-2022 09:20   conf/logback.xml
    17012  06-06-2022 09:20   conf/graphdb.properties
      512  06-06-2022 09:20   conf/graphdb.license
      960  06-06-2022 09:20   conf/tools-logback.xml
     1176  06-06-2022 09:20   repositories/repo/owlim.properties
   695047  06-06-2022 09:20   jcmd_histogram.txt
    52719  06-06-2022 09:20   thread_dump.txt
    25280  06-06-2022 09:20   system.properties
     1996  06-06-2022 09:20   system.ttl
---------                     -------
  1202903                     33 files
```

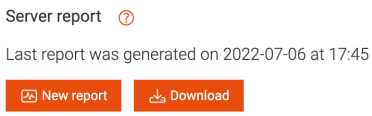
In a cluster, the report can be run from each node in the group. It adds the following to the standard report:

- Report data for each node in the cluster
- Information about the cluster status in the `graphdb-server-report-<timestamp>/cluster/cluster-status.json` file: endpoints, status, and state of each node
- `cluster-config.ttl`: The cluster configuration in Turtle format

Each node where a report is requested triggers the report for other nodes, and waits for the result until it is ready, or until a certain time limit is reached. The maximum time to wait is configured via the `graphdb.wait.report.minutes property` in the `graphdb.properties` with a default of 120 minutes. In case of a timeout or an error, it will be written to the `info.txt` file for the corresponding node.

Create report from the Workbench

Go to *Help* ▸ *System information*. Click on *New report* in the *Application info* tab to obtain a new one, wait until it is ready, and download it. It is downloaded in `.zip` format as `graphdb-server-report-<timestamp>`.



Create report through the report script

The `generate-report` script can be found in the `bin` folder in the GraphDB distribution. It needs `graphdb-pid` — the GraphDB for which you want a report. An optional argument is `output-file`, the default for which is `graphdb-server-report.zip`.

6.4.7.2 Logs

GraphDB uses slf4j for logging through the Logback implementation (the RDF4J facilities for log configuration discovery are no longer used). Instead, the whole distribution has a central place for the `logback.xml` configuration file in *GraphDB-HOME/conf/logback.xml*. If you use the `.war` file setup, you can provide the log file location through a system parameter, or GraphDB will pick it up from the generated `.war` file.

Note: Check the Logback configuration location rules for more information.

On startup, GraphDB logs the logback configuration file location:

```
[INFO ] 2022-06-06 09:44:26,179 [main | c.o.g.Config] Using 'file:/opt/graphdb/conf/logback.xml' as
↪logback's configuration file
```

Setting up the root logger

The default `root` logger is set to `info`. You can change it in several ways:

- Edit the `logback.xml` configuration file.

  Note: You do not have to restart the database as it will check the file for changes every 30 seconds, and will reconfigure the logger.

- Change the log level through the logback JMX configurator. For more information, see the Logback manual chapter 10.
- Start each component with `graphdb.logger.root.level` set to your desired root logging level. For example:

  ```
  bin/graphdb -Dgraphdb.logger.root.level=WARN
  ```

Logs location

By default, all database components and tools log in *GraphDB-HOME/logs* when run from the `bin` folder. If you set up GraphDB by deploying `.war` files into a standalone servlet container, the following rules apply:

1. To log in a specified directory, set the `logDestinationDirectory` system property.
2. If GraphDB is run in Tomcat, the logs can be found in `$catalina.base/logs/graphdb`.
3. If GraphDB is run in Jetty, the logs can be found in `$jetty.base/logs/graphdb`.
4. Otherwise, all logs are in the `logs` subdirectory of the current working directory for the process.

Log files

Different information is logged in different files. This makes it easier to follow what goes on in different parts of the system.

File name	Description
`main.log`	Contains all operational and error messages coming from the main part of the engine. Provides information about your overall system state, active repositories, plugins, operations.
`error.log`	Contains all ERROR messages returned while the instance was running. The error messages are also present, usually with more context, in the main log. The error log is useful mostly in cases where the main log moves too fast to examine the main log.
`query.log`	Contains all queries that were sent to the database. The format is machine-readable and allows you to replay the queries when debugging a problem. All file inserts would leave a trace somewhere, unless deleted explicitly, but an update query would be invisible to anyone but the person executing it. If this log is set to debug level, it will store all queries.
`slow-query.log`	Contains slow queries — queries that take more than a set amount of seconds. This is controlled by the `graphdb.engine.log-slow-queries-time` parameter and defaults to 60.
`audit.log`	This log only appears on a running GraphDB instance with security ON. Contains information about each HTTP request — from which IP address did it come from, with which user, what were the contents (truncated), what was the response code. To start an audit trail, at least one of the `graphdb.audit.role` and `graphdb.audit.repository` parameters should be enabled.

The main log is the log that is usually examined when monitoring for problems. All recent logs are included in the server report to facilitate reporting issues.

You can configure the logs with the `logback.xml` file. By default, this is located inside the `conf` directory, together with the GraphDB properties file. When starting GraphDB from `bin/graphdb`, it prints a message to the console with information about where it sources its logging configurations.

The `logback.xml` file is heavily commented with instructions about what options you can set and how to set them. Entries in this file let you set parameters such as:

- The maximum age or size of a log file before it is archived
- The format of log file names, such as whether and where to include the system date
- What types of things get logged and what level of `INFO`/`WARN`/`ERROR` severity should trigger log entries
- Level of verbosity for the different loggers

6.5 Docker and Helm

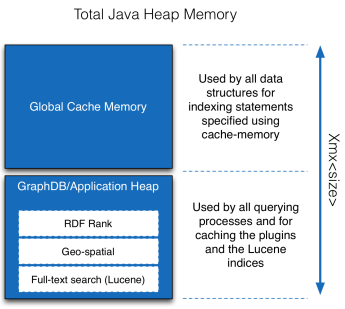
Run GraphDB in a Docker container: If you use Docker and containers, we provide ready-to-use Docker images.

Run GraphDB with Helm charts: From version 9.8 onwards, GraphDB can be deployed with open-source Helm charts. See how to set up complex GraphDB deployments on Kubernetes.

6.6 Configuring GraphDB memory

6.6.1 Configure Java heap memory

The following diagram offers a view of the memory use by the GraphDB structures and processes:



To specify the maximum amount of heap space used by a JVM, use the `-Xmx` virtual machine parameter.

6.6.2 Single global page cache

GraphDB's cache strategy, the **single global page cache**, employs the concept of one global cache shared between all internal structures of all repositories. This way, you no longer have to configure the `cache-memory`, `tuple-index-memory`, and `predicate-memory`, or size every repository and calculate the amount of memory dedicated to it. If at a given moment one of the repositories is being used more, it will naturally get more slots in the cache.

The global page cache size is dynamic and is determined by the given `-Xmx` value. It is set as follows:

Heap size	Global page cache size
Less than 4GB	25%
4-8GB	Linear, starting at 25% and ending at 30%
8-16GB	Linear, starting at 30% and ending at 35%
16-32GB	Linear, starting at 35% and ending at 40%
32-100GB	40%
Over 100GB	Max value of 40GB

The current global page cache size can be set manually by specifying: `-Dgraphdb.page.cache.size=3G`.

If you do not specify `graphdb.page.cache.size`, it will be determined by the heap range as outlined above.

Warning: Unlike the *entity pool memory* described below, which is configurable, the single global page cache is exclusively an on-heap structure.

Note: You do not have to change/edit your repository configurations. The new cache will be used when you upgrade to the new version.

6.6.3 Configure entity pool memory

By default, all entity pool structures are residing on-heap, i.e., inside the regular JVM heap. The `graphdb.engine.onheap.allocation` property is used to configure memory allocation not only for the entity pool but also for the other structures. It also specifies the entity pool on-heap allocation regardless of whether the deprecated property `graphdb.epool.onheap` is set to `true`.

Note: To activate the old behavior, i.e., the entity pool residing off-heap, you can enable off-heap allocation with `-Dgraphdb.epool.onheap=false`.

If you are concerned that the process will eat up an unlimited amount of memory, you can specify a maximum size with `-XX:MaxDirectMemorySize`, which defaults to the `-Xmx` parameter (at least in OpenJDK and Oracle JDK).

6.6.4 Sample memory configuration

This is a sample configuration demonstrating how to correctly size a GraphDB server with a single repository. The loaded dataset is estimated to 500 million RDF statements and 150 million unique entities. As a rule of thumb, the average number of unique entities compared to the total number of statements in a standard dataset is 1:3.

Configuration parameter	Description	Example value
Total OS memory	Total physical system memory.	16 GB
On-heap JVM (-Xmx) configuration	Maximum heap memory allocated by the JVM process.	10 GB
`graphdb.page.cache.size`	Global single cache shared between all internal structures of all repositories.	`5g`
Remaining on-heap memory for query execution	Raw estimate of the memory for query execution. A higher value is required if many, long running analytical queries are expected.	~4.5 GB
`entity-index-size` ("Entity index size") stored on-heap by default **Warning:** This parameter should not exceed the `Integer.MAX_VALUE` of 2,147,483,647.	Size of the initial entity pool hash table. The recommended value is equal to the total number of unique entities.	`150000000`
Memory footprint of the entity pool stored on-heap by default	Calculated from `entity-index-size` and total number of entities. This memory will be taken after the repository initialization.	~2.5 GB
Remaining OS memory	Raw estimate of the memory left to the OS.	~3.5 GB

6.6.5 Setting a tupleset/distinct memory threshold

In order to make sure that no OutOfMemoryExceptions are thrown while working with an active GraphDB repository, you need to set an upper bound value for the memory consumed by all instances of the `tupleSet/distinct` collections. This is done with the `-Dgraphdb.query.memory.threshold` parameter, whose default value is 250m and can be changed. If this value is surpassed, a QueryEvaluationException is thrown so as to avoid running out of memory due to hungry `distinct/group` by operation.

6.7 Performance optimizations

The best performance is typically measured by the shortest load time and the fastest query answering. Here are all the factors that affect GraphDB performance:

- *Configuring GraphDB memory*
- *Data loading and index optimizations*
  - *Dataset loading*
  - *Optional GraphDB indexes*
  - *Cache and index monitoring and optimizations*
- *Query optimizations*
- *Query profiling with the Explain plan*
- *Inference optimizations*
  - *Delete optimizations*
  - *Rules optimizations*
  - *Optimization of owl:sameAs*
  - *RDFS and OWL support optimizations*

6.7.1 Data loading and index optimizations

The life cycle of a repository instance typically starts with the initial loading of datasets, followed by the processing of queries and updates. The loading of a large dataset can take a long time — up to 12 hours for one billion statements with inference. Therefore, during loading, it is often helpful to use a different configuration than the one for a normal operation.

Furthermore, if you frequently load a certain dataset, since it gradually changes over time, the loading configuration can evolve as you become more familiar with the GraphDB behavior towards this dataset. Many dataset properties only become apparent after the initial load (such as the number of unique entities) and this information can be used to optimize the loading step for the next round or to improve the configuration for a normal operation.

6.7.1.1 Dataset loading

The following is a typical initialization life cycle:

1. *Configure a repository* for best loading performance with many estimated parameters.
2. Load data.
3. Examine dataset properties.
4. Refine loading configuration.
5. Reload data and measure improvement.

Unless the repository has to handle queries during the initialization phase, it can be configured with the minimum number of options and indexes:

```
enablePredicateList = false (unless the dataset has a large number of predicates)
enable-context-index = false
in-memory-literal-properties = false
```

Normal operation

The size of the data structures used to index entities is directly related to the number of unique entities in the loaded dataset. These data structures are always kept in memory. In order to get an upper bound on the number of unique entities loaded and to find the actual amount of RAM used to index them, it is useful to know the contents of the storage folder.

The total amount of memory needed to index entities is equal to the sum of the sizes of the files `entities.index` and `entities.hash`. This value can be used to determine how much memory is used and therefore how to divide the remaining memory between the cache memory, etc.

An upper bound on the number of unique entities is given by the size of `entities.hash` divided by 12 (memory is allocated in pages and therefore the last page will likely not be full).

The `entities.index` file is used to look up entries in the file `entities.hash`, and its size is equal to the value of the `entity-index-size` parameter multiplied by 4. Therefore, the `entity-index-size` parameter has less to do with efficient use of memory and more with the performance of entity indexing and lookup. The larger this value, the less collisions occur in the `entities.hash` table. A reasonable size for this parameter is at least half the number of unique entities. However, the size of this data structure is never changed once the repository is created, so this knowledge can only be used to adjust this value for the next clean load of the dataset with a new (empty) repository.

The following parameters can be adjusted:

Parameter	Description
`entity-index-size` *(see more)*	Set to a large enough value.
`enablePredicateList` *(see more)*	Can speed up queries (and loading).
`enable-context-index` *(see more)*	Provides better performance when executing queries that use contexts.
`in-memory-literal-properties` *(see more)*	Defines whether to keep the properties of each literal in-memory.

Furthermore, the inference semantics can be adjusted by choosing a different ruleset. However, this will require a reload of the whole repository, otherwise some inferences may remain in the wrong location.

Note: The optional indexes can be built at a later point when the repository is used for query answering. You need to experiment using typical query patterns from the user environment.

6.7.1.2 Optional GraphDB indexes

Predicate lists

Predicate lists are two indexes (`SP` and `OP`) that can improve performance in the following situations:

- When loading/querying datasets that have a large number of predicates;
- When executing queries or retrieving statements that use a wildcard in the predicate position, e.g., the statement pattern: `dbpedia:Human ?predicate dbpedia:Land`.

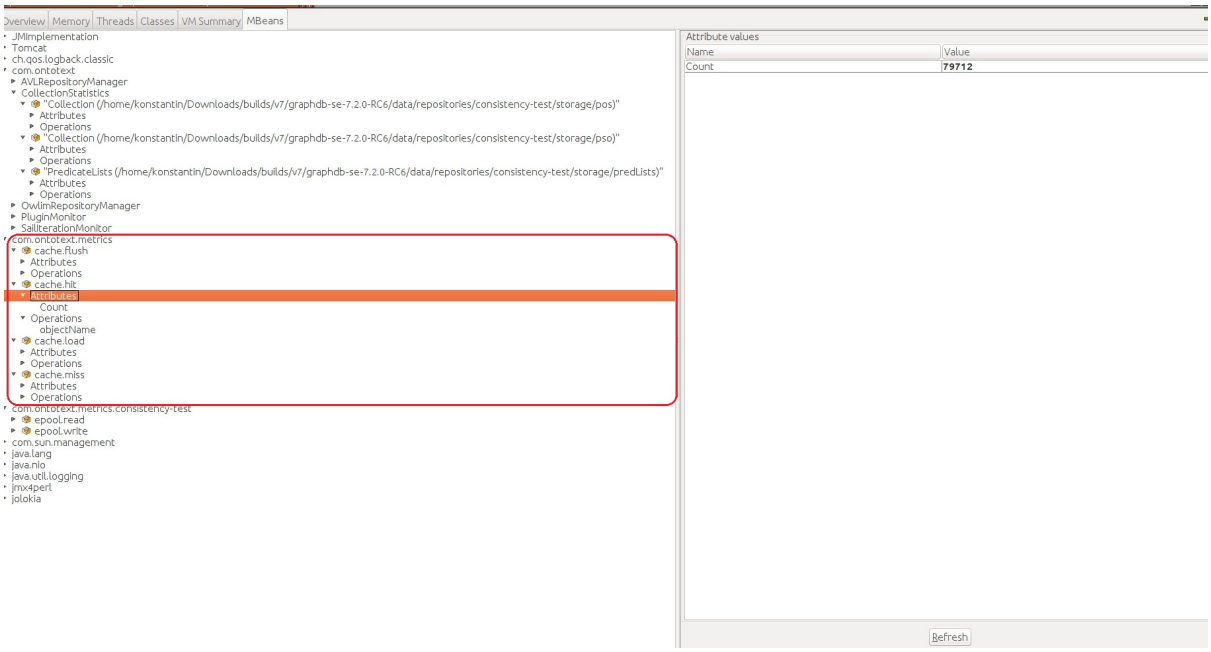
As a rough guideline, a dataset with more than about 1,000 predicates will benefit from using these indexes for both loading and query answering. Predicate list indexes are not enabled by default, but can be switched on using the `enablePredicateList` configuration parameter.

Context index

To provide better performance when executing queries that use contexts, you can use the context index `CPSO`. It is enabled by using the `enable-context-index` configuration parameter.

6.7.1.3 Cache and index monitoring and optimizations

Statistics are kept for the main index data structures, and include information such as cache hits/misses, file reads/writes, etc. This information can be used to fine-tune GraphDB memory configuration, and can be useful for 'debugging' certain situations, such as understanding why load performance changes over time or with particular datasets.



For each index, there will be a `CollectionStatistics` MBean published, which shows the cache and file I/O values updated in real time:

Package	com.ontotext
MBean name	`CollectionStatistics`

The following information is displayed for each MBean/index:

Attribute	Description
`CacheHits`	The number of operations completed without accessing the storage system.
`CacheMisses`	The number of operations completed, which needed to access the storage system.
`FlushInvocations`	
`FlushReadItems`	
`FlushReadTimeAverage`	
`FlushReadTimeTotal`	
`FlushWriteItems`	
`FlushWriteTimeAverage`	
`FlushWriteTimeTotal`	
`PageDiscards`	The number of times a non-dirty page's memory was reused to read in another page.
`PageSwaps`	The number of times a page was written to the disk, so its memory could be used to load another page.
`Reads`	The total number of times an index was searched for a statement or a range of statements.
`Writes`	The total number of times a statement was added to a collection.

The following operations are available:

Operation	Description
`resetCounters`	Resets all the counters for this index.

Ideally, the system should be configured to keep the number of cache misses to a minimum. If the ratio of hits to misses is low, consider increasing the memory available to the index (if other factors permit this).

Page swaps tend to occur much more often during large scale data loading. Page discards occur more frequently during query evaluation.

6.7.1.4 Index compacting

In some cases, database indexes get fragmented over time and with the accumulation of updates. This may lead to a slowdown in data import.

Index compacting is a useful method to tackle this. To enable it, run:

```
INSERT DATA {
    [] <http://www.ontotext.com/compactIndexes> [] .
}
```

This will:

1. Shut down the repository internally.
2. Scan the indexes.
3. Rebuild them.
4. Reinitialize the repository.

Warning: Index compacting is only suitable for specific cases.

6.7.2 Query optimizations

GraphDB uses a number of query optimization techniques by default. They can be disabled by using the `enable-optimization` configuration parameter set to `false`, however there is rarely any need to do this.

See also:

GraphDB's *Query profiling with the Explain plan* explains how to view query plans and applied optimizations.

Note: GraphDB allows hints for guiding the query optimizer.

6.7.2.1 Caching literal language tags

This optimization applies when the repository contains a large number of literals with language tags, and it is necessary to execute queries that filter based on language, for example, using the following SPARQL query construct:

```
FILTER ( langMatches(lang(?name), "es") )
```

In this case, the `in-memory-literal-properties` configuration parameters can be set to `true`, causing the data values with language tags to be cached.

6.7.2.2 Not enumerating sameAs

During query answering, all URIs from each equivalence class produced by the *sameAs optimization* are enumerated. You can use the `onto:disable-sameAs` pseudo-graph to significantly reduce these duplicate results by returning a single representative from each equivalence class.

See also:

See *Other special query behavior*.

Consider these example queries executed against the FactForge combined dataset. Here, the default is to enumerate:

```
PREFIX onto: <http://www.ontotext.com/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
SELECT *
# FROM onto:disable-sameAs
{
    ?x a dbo:Airport; dbo:location dbr:Bulgaria
}
```

As shown, this query returns 30 results. If you uncomment the `FROM onto:disable-sameAs` line, the query returns 27 results.

The *Expand results over equivalent URIs* checkbox in the GraphDB Workbench SPARQL editor plays a similar role, but the meaning is reversed.

Warning: If the query uses a filter over the textual representation of a URI, such as `filter(strstarts(str(?x),"http://dbpedia.org/ontology"))`, the response may omit some valid solutions, as not all URIs within an equivalence class are matched against the filter.

6.7.3 Inference optimizations

6.7.3.1 Delete optimizations

GraphDB's *inference* policy is based on materialization, where implicit statements are inferred from explicit statements as soon as they are inserted into the repository, using the specified semantics ruleset. This approach has the advantage of achieving query answering very quickly, since no inference needs to be done at query time.

However, no justification information is stored for inferred statements, therefore deleting a statement normally requires a full re-computation of all inferred statements. This can take a very long time for large datasets.

GraphDB uses a special technique for handling the deletion of explicit statements and their inferences, called *smooth delete*. It allows fast delete operations as well as ensures that schemas can be changed when necessary.

The algorithm

The algorithm for identifying and removing the inferred statements that can no longer be derived by the explicit statements that have been deleted, is as follows:

1. Use forward chaining to determine what statements can be inferred from the statements marked for deletion.
2. Use backward chaining to see if these statements are still supported by other means.
3. Delete explicit statements and the no longer supported inferred statements.

Note: We recommend that you mark the visited statements as read-only. Otherwise, as almost all delete operations follow inference paths that touch schema statements, which then lead to almost all other statements in the repository, the smooth delete can take a very long time. However, since a read-only statement cannot be deleted, there is no reason to find what statements are inferred from it (such inferred statements might still get deleted, but they will be found by following other inference paths).

Statements are marked as read-only if they occur in the `Axioms` section of the `ruleset` files (standard or custom) or are loaded at initialization time via the *imports configuration parameter*.

Note: When using smooth delete, we recommend that you load all ontology/schema/vocabulary statements using the `imports` configuration parameter.

Example

Consider the following statements:

```
Schema:
<foaf:name> <rdfs:domain> <owl:Thing> .
<MyClass> <rdfs:subClassOf> <owl:Thing> .

Data:
<wayne_rooney> <foaf:name> "Wayne Rooney" .
<Reviewer40476> <rdf:type> <MyClass> .
<Reviewer40478> <rdf:type> <MyClass> .
<Reviewer40480> <rdf:type> <MyClass> .
<Reviewer40481> <rdf:type> <MyClass> .
```

When using the `owl-horst` ruleset, the removal of the statement:

```
<wayne_rooney> <foaf:name> "Wayne Rooney"
```

will cause the following sequence of events:

```
rdfs2:
x a y - (x=<wayne_rooney>, a=foaf:name, y="Wayne Rooney")
a rdfs:domain z (a=foaf:name, z=owl:Thing)
-----------------------
x rdf:type z  - The inferred statement [<wayne_rooney> rdf:type owl:Thing] is to be removed.
```

```
rdfs3:
x a u - (x=<wayne_rooney>, a=rdf:type, u=owl:Thing)
a rdfs:range z (a=rdf:type, z=rdfs:Class)
-----------------------
u rdf:type z - The inferred statement [owl:Thing rdf:type rdfs:Class] is to be removed.
```

```
rdfs8_10:
x rdf:type rdfs:Class - (x=owl:Thing)
-----------------------
x rdfs:subClassOf x - The inferred statement [owl:Thing rdfs:subClassOf owl:Thing] is to be removed.
```

```
proton_TransitiveOver:
y q z - (y=owl:Thing, q=rdfs:subClassOf, z=owl:Thing)
p protons:transitiveOver q - (p=rdf:type, q=rdfs:subClassOf)
x p y - (x=[<Reviewer40476>, <Reviewer40478>, <Reviewer40480>, <Reviewer40481>], p=rdf:type,
↪y=owl:Thing)
-----------------------
x p z - The inferred statements [<Reviewer40476> rdf:type owl:Thing], etc., are to be removed.
```

Statements such as `[<Reviewer40476> rdf:type owl:Thing]` exist because of the statements `[<Reviewer40476> rdf:type <MyClass>]` and `[<MyClass> rdfs:subClassOf owl:Thing]`.

In large datasets, there are typically millions of statements `[X rdf:type owl:Thing]`, and they are all visited by the algorithm.

The `[X rdf:type owl:Thing]` statements are not the only problematic statements considered for removal. Every class that has millions of instances leads to similar behavior.

One check to see if a statement is still supported requires about 30 query evaluations with OWL-Horst, hence the slow removal.

If `[owl:Thing rdf:type owl:Class]` is marked as an axiom (because it is derived by statements from the schema, which must be axioms), then the process stops when reaching this statement. So, the schema (the system statements) must necessarily be imported through the `imports` configuration parameter in order to mark the schema statements as axioms.

Schema transactions

As mentioned above, ontologies and schemas imported at initialization time using the *imports configuration parameter* configuration parameter are flagged as read-only. However, there are times when it is necessary to change a schema. This can be done inside a 'system transaction'.

The user instructs GraphDB that the transaction is a system transaction by including a dummy statement with the special `schemaTransaction` predicate, i.e.:

```
_:b1 <http://www.ontotext.com/owlim/system#schemaTransaction> _:b2
```

This statement is not inserted into the database, but is rather serving as a flag telling GraphDB that the statements from this transaction are going to be inserted as read-only; all statements derived from them are also marked as read-only. When you delete statements in a system transaction, you can remove statements marked as read-only, as well as statements derived from them. Axiom statements and all statements derived from them stay untouched.

6.7.3.2 Rules optimizations

GraphDB includes the useful feature of rule optimizing that allows you to profile and debug rule performance.

How to enable rule profiling

Rule profiling prints out statistics about rule execution.

To enable rule profiling, start GraphDB with the following Java option:

```
-Denable-debug-rules=true
```

This enables the collection of rule statistics (various counters).

> **Warning: Rule Profiling Limitations**
>
> - Must use a **custom ruleset**, since built-in rulesets do not have the required instrumentation (counters);
> - The debug rules statistics are available only for importing data in **serial mode**. It does not work for Parallel Inferencing, which is default. Check *Force serial pipeline* in the *Import settings* dialog to enable it.

> **Warning:** Rule profiling slows down the rule execution (the leading premise checking part) by 10-30%, so do not use it in production.

Log file

When rule profiling is enabled:

- Complete rule statistics are printed at every million statements, every 5 minutes, or on shutdown, depending on which occurs first.
- They are written to `graphdb-folder/logs/main-<date>.log`;
- The descriptive rule stats format looks like this:

```
----------rs start----------
Rule statistics for repository <name> :
RULE: ...

Time overall (all rules): ... ns.
----------rs end----------
```

- A tabular format of the descriptive rule stats is also supported, e.g.:

```
----------rs csv start----------
Repository name: university
RULE;TIME;INVOKED;ITERNEXTS;FIRED;FIRETIME;INFERRED
prp_spo1_1;36973700;66;81;81;36917600;28
prp_spo1_0;34354600;17;83;83;34310500;25
----------rs csv end----------
```

- Stats are printed for each active repository.
- Stats are cumulative, so find the last section `rs start` ... `rs end` for your repo of interest.
- Rule variants are ordered by total time (descending).

For example, consider the following rule:

```
Id: ptop_PropRestr
  t <ptop:premise>     p
  t <ptop:restriction> r
  t <ptop:conclusion>  q
  t <rdf:type>         <ptop:PropRestr>
  x p y
  x r y
  ----------------
  x q y
```

This is a conjunction of two props. It is declared with the axiomatic (A-Box) triples involving `t`. Whenever the premise `p` and restriction `r` hold between two resources, the rule infers the conclusion `q` between the same resources, i.e., `p & r => q`.

The corresponding log for variant 4 of this rule may look like the following:

```
RULE ptop_PropRestr_4 invoked 163,475,763 times.
ptop_PropRestr_4:
e b f
a ptop_premise b
a rdf_type ptop_PropRestr
e c f
a ptop_restriction c
a ptop_conclusion d
------------------------------------
e d f

a ptop_conclusion d invoked 1,456,793 times and took 1,814,710,710 ns.
a rdf_type ptop_PropRestr invoked 7,261,649 times and took 9,409,794,441 ns.
a ptop_restriction c invoked 1,456,793 times and took 1,901,987,589 ns.
e c f invoked 17,897,752 times and took 635,785,943,152 ns.
a ptop_premise b invoked 10,175,697 times and took 9,669,316,036 ns.
Fired 1,456,793 times and took 157,163,249,304 ns.
Inferred 1,456,793 statements.
Time overall: 815,745,001,232 ns.
```

Note: Variable names are renamed due to the compilation to Java bytecode.

Understanding the output:

- The premises are checked in the order given in `RULE`. (The premise statistics printed after the blank line are not in any particular order.)
- `Invoked` is the number of times the rule variant or specific premise was checked successfully. Tracing through the rule:
  - `ptop_PropRestr_4` checked successfully 163 million times: for each incoming triple, since the lead premise (`e b f = x p y`) is a free pattern.
  - `a ptop_premise b` checked successfully 10 million times: for each `b=p` that has an axiomatic triple involving `ptop_premise`.

    This premise was selected because it has only 1 unbound variable `a` and it is first in the rule text.
  - `a rdf_type ptop_PropRestr` checked successfully 7 million times: for each `ptop_premise` that has type `ptop_PropRestr`.

    This premise was selected because it has 0 unbound variables (`after` the previous premise binds `a`).
- The time to check each premise is printed in `ns`.
- `Fired` is the number of times all premises matched, so the rule variant was fired.
- `Inferred` is the number of inferred triples.

It may be greater than `fired` if there are multiple conclusions.
It may be less than `fired` since a duplicate triple is not inferred a second time.

- `Time overall` is the total time that this rule variant took.

Excel format

The log records detailed information about each rule and premise, which is very useful when you are trying to understand which of the rules is too time-consuming. However, it can still be overwhelming because of this level of detail.

Therefore, we have developed the `rule-stats.pl` script that outputs a TSV file such as the following:

rule	ver	tried	time	patts	checks	time	fired	time	triples	speed
ptop_PropChain	4	163475763	776.3	5	117177482	185.3	15547176	590.9	9707142	12505

Parameters:

Parameter	Description
`rule`	the rule ID (name)
`ver`	the rule version (variant) or “T” for overall rule totals
`tried, time`	the number of times the rule/variant was tried, the overall time in `sec`
`patts`	the number of triple patterns (premises) in the rule, not counting the leading premise
`checks, time`	the number of times premises were checked, time in `sec`
`fired`	the number of times all premises matched, so the rule was fired
`triples`	the number of inferred triples
`speed`	inference speed, triples/sec

Run the script the following way:

```
perl rule-stats.pl main-2014-07-28.log > main-2014-07-28.xls
```

Investigating performance

The following is an example of using the Excel format to investigate where time is spent during rule execution.

Download the `time-spent-during-rule.xlsx` example file, and use it as a template.

rule	v	tried	time	pat	checks	time	fired	time	triples	speed
rdfs7_subPropertyOf	T	163476295	1999.4	2	191323783	205.8	27848020	1793.5	24903005	12455.5
ptop_PropChain	T	326951600	1185.1	30	219164757	287.5	22874870	897.5	13813732	11656.6
ptop_PropRestr	T	326951561	1184.3	16	78654527	1023.6	2869625	160.7	1457031	1230.3
rdfs2_domain	T	130143111	533.7	1	49558299	76.5	49558299	457.1	25272752	47357.1
ptop_PropChainRestr	T	490427353	482.7	31	148702055	205.7	5365740	277.1	5365740	11115.8
rdfs3_range	T	163475973	304.1	2	87483795	102.5	87483795	201.6	26317789	86531.8
ptop_TypeRestr	T	163475799	201.4	17	36478117	32.1	1469392	169.3	1469382	7296.6
rdf_type	T	33332899	119.6	2	42522646	52.9	9189831	66.7	6861382	57354.7
ptop_PropChainType2	T	360284446	80.1	31	78128145	79.5	31172	0.6	30074	375.5
owl_inverseOf	T	163475946	0.3	2	64792	0.1	64792	0.3	23746	73592.9
owl_SymPropByInvOf	T	27	0				27	0	0	0
owl_invOfBySymProp	T	27	0				27	0	27	14548.9
ptop_PropChainByPropertyChainAxiom	T	44289	0	9	11109	0	32	0	32	2108.6
ptop_TransPropAsChain	T	7	0				28	0	28	4555
rdfs12_member	T	1	0				1	0	1	10133.1
rdfs13_subClass_Literal	T	9	0				9	0	9	9986.6
rdfs6_subPropertyReflective	T	291	0				291	0	291	17319.9
rdfs8_10_subClassReflective	T	76	0				76	0	76	10138.1
			6090.7						105515097	17323.97

Note: These formulas are dynamic, and they are updated every time you change the filters.

To perform your investigation:

1. Open the results in Excel.
2. Set a filter “ver=T”, first looking at the rules in their entirety instead of rule variants.
3. Sort by “time” (fourth column) in descending order.
4. Check which rules are highlighted in red (those that take significantly long and whose speed is substantially lower than average).
5. Pick a rule (for example, `PropRestr`).
6. Filter by “rule=PropRestr” and “ver<>T” to see its variants.

rule	v	tried	time	pat	checks	time	fired	time	triples	speed
ptop_PropRestr	4	163475763	815.7	5	38248684	658.6	1456793	157.2	1456793	1785.8
ptop_PropRestr	5	163475763	368.5	5	40405825	365	1412832	3.6	238	0.6
ptop_PropRestr	2	23	0	3	9	0	0	0	0	0
ptop_PropRestr	0	5	0	1	3	0	0	0	0	0
ptop_PropRestr	1	4	0	2	6	0	0	0	0	0
ptop_PropRestr	3	3	0				0	0	0	0
			1184.2						1457031	1230.393

7. Focus on a variant to investigate the reasons for its poorer time and speed performance.

In this example, the first variant you would want to investigate will be `ptop_PropRestr_5`, as it is spending 30% of the time of this rule, and has very low speed. The reason is that it fired 1.4 million times but produced only 238 triples, so most of the inferred triples were duplicates.

You can find the definition of this variant in the log file:

```
RULE ptop_PropRestr_5 invoked 163,475,763 times.
ptop_PropRestr_5:
e c f
a ptop_restriction c
a rdf_type ptop_PropRestr
e b f
a ptop_premise b
a ptop_conclusion d
------------------------------------
e d f
```

It is very similar to the productive variant `ptop_PropRestr_4` (see *Log file* above):

- one checks `e b f. a ptop_premise b` first,
- the other checks `e c f. a ptop_restriction c` first.

Still, the function of these premises in the rule is the same and therefore the variant `ptop_PropRestr_5` (which is checked `after` 4) is unproductive.

The most likely way to improve performance would be if you make the two premises use the same axiomatic triple `ptop:premise` (emphasizing they have the same role), and introduce a `Cut`:

```
Id: ptop_PropRestr_SYM
  t <ptop:premise>     p
  t <ptop:premise>     r
  t <ptop:conclusion>  q
  t <rdf:type>         <ptop:PropRestr>
  x p y
  x r y                [Cut]
  ----------------
  x q y
```

The `Cut` eliminates the rule variant with `x r y` as leading premise. It is legitimate to do this, since the two variants are the same, up to substitution `p<->r`.

Note: Introducing a `Cut` in the original version of the rule would **not** be legitimate:

```
Id: ptop_PropRestr_CUT
  t <ptop:premise>     p
  t <ptop:restriction> r
  t <ptop:conclusion>  q
  t <rdf:type>         <ptop:PropRestr>
  x p y
  x r y                [Cut]
  ----------------
  x q y
```

since it would omit some potential inferences (in the case above, 238 triples), changing the semantics of the rule (see the example below).

Assume these axiomatic triples:

```
:t_CUT a ptop:PropRestr; ptop:premise :p; ptop:restriction :r; ptop:conclusion :q. # for ptop_PropRestr_
↪CUT
:t_SYM a ptop:PropRestr; ptop:premise :p; ptop:premise     :r; ptop:conclusion :q. # for ptop_PropRestr_
↪SYM
```

Now consider a sequence of inserted triples `:x :p :y. :x :r :y.`

- `ptop_PropRestr_CUT` **will not** infer `:x :q :y`, since no variant is fired by the second incoming triple `:x :r :y`: it is matched against `x p y`, but there is no axiomatic triple `t ptop:premise :r`.
- `ptop_PropRestr_SYM` **will** infer `:x :q :y`, since the second incoming triple `:x :r :y` will match `x p y` and `t ptop:premise :r`, then the previously inserted `:x :p :y` will match `t ptop:premise :p` and the rule will fire.

Tip: Rule execution is often non-intuitive, therefore we recommend that you detail the speed history and compare the performance after each change.

Hints on optimizing GraphDB's rulesets

The complexity of the ruleset has a significant effect on the loading performance, the number of inferred statements, and the overall size of the repository after inferencing. The complexity of the standard rulesets increases as follows:

- no inference (lowest complexity, best performance)
- RDFS-Optimized
- RDFS
- RDFS-Plus-Optimized
- RDFS-Plus
- OWL-Horst-Optimized
- OWL-Horst
- OWL-Max-Optimized
- OWl-Max
- OWL2-QL-Optimized
- OWL2-QL

- OWL2-RL-Optimized
- OWL2-RL (highest complexity, worst performance)

It needs to be mentioned that OWL-RL and OWL-QL do a lot of heavy work that is often not required by applications. For more details, see *OWL compliance*.

Know what you want to infer

Check the expansion ratio (total/explicit statements) for your dataset in order to get an idea of whether this is the result that you are expecting. For example, if your ruleset infers 4 times more statements over a large number of explicit statements, this will take time regardless of the ways in which you try to optimize the rules.

Minimize the number of rules

The number of rules and their complexity affects inferencing performance, even for rules that never infer any new statements. The reason for this is that every incoming statement is passed through every variant of every rule to check whether something can be inferred. This often results in many checks and joins, even if the rule never fires.

So, start with a minimal ruleset and only add the rules that you need. The default ruleset (RDFS-Plus-Optimized) works for many users, but you might even consider starting from RDFS. For example, if you need `owl:Symmetric` and `owl:inverseOf` on top of RDFS, you can copy only these rules from OWL-Horst to RDFS and leave out the rest.

Conversely, you can start with a bigger standard ruleset and remove the rules that you do not need.

Note: To deploy a custom ruleset, set the *ruleset configuration parameter* to the full pathname of your custom `.pie` file.

Write your rules carefully

- Be careful with the recursive rules as they can lead to an explosion in the number of inferred statements.
- Always check your spelling:
  - A misspelled variable in a premise leads to a Cartesian explosion (variables quickly growing to an intractable level) of the number of triple joins to be considered by the rule.
  - A misspelled variable in a conclusion (or the use of an unbound variable) leads to the creation of new blank nodes. This is almost never what you really want.
- Order premises by specificity. GraphDB first checks premises with the least number of unbound variables. But if there is a tie, it follows the order given by you. Since you may know the cardinalities of triples in your data, you may be in a better position to determine which premise has better specificity (selectivity).
- Use `Cut` for premises that have the same role (for an example, see *Investigating performance*), but be careful not to remove any necessary inferences by mistake.

Avoid duplicate statements

Avoid inserting explicit statements in a named graph if the same statements are inferable. GraphDB always stores inferred statements in the default graph, so this will lead to duplicating statements. This will increase the repository size and slow down query answering.

You can eliminate duplicates from query results using `DISTINCT` or `FROM onto:skip-redundant-implicit` (see *Other query behavior customizations*). However, these are slow operations, so it is better not to produce duplicate statements in the first place.

Know the implications of ontology mapping

People often use `owl:equivalentProperty`, `owl:equivalentClass` (and less often `rdfs:subPropertyOf`, `rdfs:subClassOf`) to map ontologies. However, every such assertion means that many more statements are inferred (`owl:equivalentProperty` works as a pair of `rdfs:subPropertyOf`, and `owl:equivalentClass` works as a pair of `rdfs:subClassOf`).

A good example is DCTerms (DCT): almost each DC property has a declared DCT subproperty and there is also a hierarchy amongst DCT properties, for instance:

```
dcterms:created rdfs:subPropertyOf dc:date, dcterms:date.
dcterms:date rdfs:subPropertyOf dc:date.
```

This means that every `dcterms:created` statement will expand to 3 statements. So, do not load the DC ontology unless you really need these inferred `dc:date`.

Consider avoiding inverse statements

Inverse properties (e.g., `:p owl:inverseOf :q`) offer some convenience in querying, but are never necessary:

- SPARQL natively has bidirectional data access: instead of `?x :q ?y`, you can always query for `?y :p ?x`.
- You can even invert the direction in a property path: instead of `?x :p1/:q ?y`, use `?x :p1/(^:p) ?y`.

If an ontology defines inverses but you skip inverse reasoning, you have to check which of the two properties is used in a particular dataset, and write your queries carefully.

The Provenance Ontology (PROV-O) has considered this dilemma thoroughly, and has abstained from defining inverses to *"avoid the need for OWL reasoning, additional code, and larger queries"* (see http://www.w3.org/TR/prov-o/#inverse-names).

Consider avoiding long transitive chains

A chain of N transitive relations (e.g., `rdfs:subClassOf`) causes GraphDB to infer and store a further $(n^2 - n)/2$ statements. If the relationship is also symmetric (e.g., in a family ontology with a predicate such as `relatedTo`), then there will be $n^2 - n$ inferred statements.

Consider removing the transitivity and/or symmetry of relations that make long chains. Or, if you must have them, consider the implementation of *TransitiveProperty through step property*, which can be faster than the standard implementation of `owl:TransitiveProperty`.

Consider specialized property constructs

While OWL2 has very powerful class constructs, its property constructs are quite weak. Some widely used OWL2 property constructs could be done faster.

See the document Extending OWL2 Property Constructs with OWLIM Rules for some ideas and illustrations.

Tip: To learn more, see a discussion of inferencing strategy used in a real-world setting at the Getty Research Institute.

PropChain

Consider 2-place `PropChain` instead of general `owl:propertyChainAxiom`.

`owl:propertyChainAxiom` needs to use intermediate nodes and edges in order to unroll the `rdf:List` representing the chain. Since most chains found in practice are 2-place chains (and a chain of any length can be implemented as a sequence of 2-place chains), consider a rule such as the following:

```
Id: ptop_PropChain
  t <ptop:premise1>   p1
  t <ptop:premise2>   p2
  t <ptop:conclusion> q
  t <rdf:type> <ptop:PropChain>
  x p1 y
  y p2 z
  ----------------
  x q z
```

It is used with axiomatic triples as in the following:

```
@prefix ptop: <http://www.ontotext.com/proton/protontop#>.
:t a ptop:PropChain; ptop:premise1 :p1; ptop:premise2 :p2; ptop:conclusion :q.
```

transitiveOver

`psys:transitiveOver` has been part of Ontotext's PROTON ontology since 2008. It is defined as follows:

```
Id: psys_transitiveOver
  p <psys:transitiveOver> q
  x p y
  y q z
  ---------------
  x p z
```

It is a specialized `PropChain`, where `premise1` and `conclusion` coincide. It allows you to chain `p` with `q` on the right, yielding `p`. For example, the inferencing of types along the class hierarchy can be expressed as:

```
rdf:type psys:transitiveOver rdfs:subClassOf
```

TransitiveProperty through step property

`owl:TransitiveProperty` is widely used and is usually implemented as follows:

```
Id: owl_TransitiveProperty
  p <rdf:type> <owl:TransitiveProperty>
  x p y
  y p z
  ----------
  x p z
```

You may recognize this as a self-chain, thus a specialization of `psys:transitiveOver`, i.e.:

```
?p rdf:type owl:TransitiveProperty <=> ?p psys:transitiveOver ?p
```

Most transitive properties comprise transitive closure over a basic 'step' property. For example, `skos:broaderTransitive` is based on `skos:broader` and is implemented as:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive a owl:TransitiveProperty.
```

Now consider a chain of N `skos:broader` between two nodes. The `owl_TransitiveProperty` rule has to consider every split of the chain, thus inferring the same closure between the two nodes N times, leading to quadratic inference complexity.

This can be optimized by looking for the step property `s` and extending the chain only at the right end:

```
Id: TransitiveUsingStep
  p <rdf:type> <owl:TransitiveProperty>
  s <rdfs:subPropertyOf> p
  x p y
  y s z
  ----------
  x p z
```

However, this would not make the same inferences as `owl_TransitiveProperty` if someone inserts the transitive property explicitly (which is a bad practice).

A more robust approach is to declare the step and transitive properties together using `psys:transitiveOver`, for instance:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive psys:transitiveOver skos:broader.
```

Translating OWL constructs to specialized property constructs

Other options for optimizing your rulesets to make them faster:

- `ptop:transitiveOver` is faster than `owl:TransitiveProperty`: quadratic vs cubic complexity over the length of transitive chains.
- `ptop:PropChain` (a 2-place chain) is faster than general `owl:propertyChainAxiom` (n-place chain) because it does not need to unroll the `rdf:List` underlying the representation of `owl:propertyChainAxiom`.

Under some conditions, you can translate the standard OWL constructs to these custom constructs to have both standards compliance and faster speed:

- use rule `TransitiveUsingStep`; if every `TransitiveProperty` `p` (e.g., `skos:broaderTransitive`) is defined over a step property `s` (e.g., `skos:broader`) and you do not insert `p` directly.
- if you use only 2-step `owl:propertyChainAxiom`, then translate them to custom using the following rule, and infer using rule `ptop_PropChain`:

```
Id: ptop_PropChain_from_propertyChainAxiom
  q <owl:propertyChainAxiom> l1
  l1 <rdf:first> p1
  l1 <rdf:rest> l2
  l2 <rdf:first> p2
  l2 <rdf:rest> <rdf:nil>
  ----------------------
  t <ptop:premise1>   p1
  t <ptop:premise2>   p2
  t <ptop:conclusion> q
  t <rdf:type> <ptop:PropChain>
```

Additional ruleset usage optimization

GraphDB applies special processing to the following rules so that inferred statements such as `<P a rdf:Property>`, `<P rdfs:subPropertyOf P>` and `<X a rdfs:Resource>` can appear in the repository without slowdown of inference:

```
/*partialRDFS*/
Id: rdf1_rdfs4a_4b
x a y
--------------------------------
a <rdf:type> <rdf:Property>
x <rdf:type> <rdfs:Resource>
a <rdf:type> <rdfs:Resource>
y <rdf:type> <rdfs:Resource>
/*partialRDFS*/

Id: rdfs6
a <rdf:type> <rdf:Property>
--------------------------------
a <rdfs:subPropertyOf> a
```

According to them, whatever statement comes into the repository, its subject, predicate and object are resources and its predicate is an `rdf:Property`, which then becomes `subPropertyOf` itself using the second rule (the reflexivity of `subPropertyOf`). These rules, however, if executed every time, present a similar challenge to when using `owl:sameAs`. To avoid the performance drop, GraphDB obtains these statements through code so that `<P a rdf:Property>` and `<X a rdfs:Resource>` are asserted only once — when a property or a resource is encountered for the first time (except in the 'optimized' rulesets, where `rdfs:Resource` is omitted because of the very limited use of such inference).

If we start with the `empty` ruleset, `<P a rdf:Property>`, `<P rdfs:subPropertyOf P>` and `<X a rdfs:Resource>` statements will not be inferred until we switch the ruleset. Then the inference will take place for the new properties and resources only.

Inversely, if we start with a non-empty ruleset and switch to the `empty` one, then the statements `<P a rdf:Property>`, `<P rdfs:subPropertyOf P>` and `<X a rdfs:Resource>` inferred so far will remain. This is true even if we delete statements or recompute the inferred closure.

6.7.3.3 Optimization of `owl:sameAs`

The OWL same as optimization uses the OWL `owl:sameAs` property to create an equivalence class between nodes of an RDF graph. An equivalence class has the following properties:

- Reflexivity, i.e., A -> A
- Symmetricity, i.e., if A -> B then B -> A
- Transitivity, i.e., if A -> B and B -> C then A -> C

Instead of using *simple rules and axioms* for `owl:sameAs` (actually 2 axioms that state that it is Symmetric and Transitive), GraphDB offers an effective non-rule implementation, i.e., the `owl:sameAs` support is hard-coded. The rules are commented out in the `.pie` files, and are left only as a reference.

In GraphDB, the equivalence class is represented with a single node, thus avoiding the explosion of all N^2 `owl:sameAs` statements, and instead storing the members of the equivalence class in a separate structure. In this way, the ID of the equivalence class can be used as an ordinary node, which eliminates the need to copy statements by subject, predicate and object. So, all these copies are replaced by a single statement.

There is no restriction on how to choose this single node that will represent the class as a whole, so we pick the first node that enters the class. After creating such a class, all statements with nodes from this class are altered to use the class representative. These statements also participate in the inference.

The equivalence classes may grow when more `owl:sameAs` statements containing nodes from the class are added to the repository. Every time you add a new `owl:sameAs` statement linking two classes, they merge into a single class.

During query evaluation, GraphDB uses a kind of backward chaining by enumerating equivalent URIs, thus guaranteeing the completeness of the inference and query results. It takes special care to ensure that this optimization does not hinder the ability to distinguish between explicit and implicit statements.

Removing `owl:sameAs` statements

When removing `owl:sameAs` statements from the repository, some nodes may remain detached from the class they belong to, the class may split into two or more classes, or may disappear altogether. To determine the behavior of the classes in each particular case, you should track what the original `owl:sameAs` statements were and which of them remain in the repository. All statements coming from the user (either through a SPARQL query or through the RDF4J API) are marked as explicit, and every statement derived from them during inference is marked as inferred. So, by knowing which the remaining explicit `owl:sameAs` statements are, you can rebuild the equivalence classes.

Note: It is not necessary to rebuild all the classes but only the ones that were referred to by the removed `owl:sameAs` statements.

When nodes are removed from classes, or when classes split or disappear, the new classes (or the removal of classes) yield new representatives. So, statements using the old representatives should be replaced with statements using the new ones. This is also achieved by knowing which statements are explicit. The representative statements (i.e., statements that use representative nodes) are flagged as a special type of statement that may cease to exist after making changes to the equivalence classes. In order to make new representative statements, you should use the explicit statements and the new state of the equivalence classes (e.g., it is not necessary to process all statements when only a single equivalence class has been changed). The representative statements, although being volatile, are visible to SPARQL queries and the inferencer, whereas the explicit statements that use nodes from the equivalence classes remain invisible and are only used for rebuilding the representative statements.

Disabling the `owl:sameAs` support

By default, the `owl:sameAs` support is enabled in all rulesets except for Empty (without inference), RDFS, and RDFS-Plus. However, disabling the `owl:sameAs` behavior may be beneficial in some cases. For example, it can save you time or you may want to visualize your data without the statements generated by `owl:sameAs` in queries or inferences of such statements.

To disable `owl:sameAs`, use:

- (for individual queries) `FROM onto:disable-sameAs` system graph;
- (for the whole repository) the `disable-sameAs` configuration parameter (Boolean, defaults to `false`). This disables all inferences.

Disabling `owl:sameAs` by query does not remove the inferences that have taken place because of `owl:sameAs`.

Consider the following example:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  <urn:A> owl:sameAs <urn:B> .
  <urn:A> a <urn:Class1> .
  <urn:B> a <urn:Class2> .
}
```

This leads to `<urn:A>` and `<urn:B>` being instances of the intersection of the two classes:

```
PREFIX test: <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  test:Intersection owl:intersectionOf (<urn:Class1> <urn:Class2>) .
}
```

If you query what instances the intersection has:

```
PREFIX test: <http://test.com/>

SELECT * {
  ?s a test:Intersection .
}
```

the response will be: `<urn:A>` and `<urn:B>`. Using `FROM onto:disable-sameAs` returns only the equivalence class representative (e.g., `<urn:A>`). But it does not disable the inference as a whole.

In contrast, when you set up a repository with the `disable-sameAs` repository parameter set to `true`, the inference `<urn:A> a :Intersection` will not take place. Then, if you query what instances the intersection has, it will return neither `<urn:A>`, nor `<urn:B>`.

Apart from this difference that affects the scope of action, disabling `owl:sameAs` both as a repository parameter and a `FROM` clause in the query will have the same behavior.

See how to configure the *Expand results over owl:sameAs* setting from the Workbench in *Customizing Workbench behavior*.

How `disable-sameAs` interferes with the different rulesets

The following parameters can affect the `owl:sameAs` behavior:

- `ruleset` — `owl:sameAs` support is disabled for the `empty` ruleset. Switching to a non-empty ruleset (e.g., owl-horst-optimized) enables the inference and if it is launched again, the results show all inferred statements, as well as the ones generated by `owl:sameAs`. They do not include any `<P a rdf:Property>` and `<X a rdfs:Resource>` statements (see *Rules optimizations*).
- `disable-sameAs: true + inference` — disables the `owl:sameAs` expansion but still shows the other implicit statements. However, these results will be different from the ones retrieved by `owl:sameAs + inference` or when there is no inference.
- `FROM onto:disable-sameAs` — including this clause in a query produces different results with different rulesets.
- `FROM onto:explicit` — using only this clause (or with `FROM onto:disable-sameAs`) produces the same results as when the inferencer is disabled (as with the `empty` ruleset). This means that the ruleset and the `disable-sameAs` parameter do not affect the results.
- `FROM onto:explicit` + `FROM onto:implicit` — produces the same results as if both clauses are omitted.
- `FROM onto:implicit` — using this clause returns only the statements derived by the inferencer. Therefore, with the `empty` ruleset, it is expected to produce no results.
- `FROM onto:implicit` + `FROM onto:disable-sameAs` — shows all inferred statements (except for the ones generated by `owl:sameAs`).

The following examples illustrate this behavior:

Example 1

If you use `owl:sameAs` with the following statements:

```
PREFIX test: <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  test:a test:b test:c .
  test:a owl:sameAs test:d .
  test:d owl:sameAs test:e .
}
```

and you want to retrieve data with this query:

```
PREFIX test: <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

DESCRIBE test:a test:b test:c test:d test:e
```

the result is the same as if you query for explicit statements when there is no inference or if you add `FROM onto:explicit`.

However, if you enable the inference, you will see a completely different picture. For example, if you use `owl-horst-optimized`, `disable-sameAs=false`, you will receive the following results:

```
test:a test:b test:c .
test:a owl:sameAs test:a .
test:a owl:sameAs test:d .
test:a owl:sameAs test:e .
test:b a rdf:Property .
test:b rdfs:subPropertyOf test:b .
test:d owl:sameAs test:a .
```

(continues on next page)

(continued from previous page)

```
test:d owl:sameAs test:d .
test:d owl:sameAs test:e .
test:e owl:sameAs test:a .
test:e owl:sameAs test:d .
test:e owl:sameAs test:e .
test:d test:b test:c .
test:e test:b test:c .
```

Example 2

If you start with the `empty` ruleset, then switch to `owl-horst-optimized`:

```
PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA {
  _:b sys:addRuleset "owl-horst-optimized" .
  _:b sys:defaultRuleset "owl-horst-optimized" .
}
```

and compute the full inference closure:

```
PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA {
  _:b sys:reinfer _:b .
}
```

the same `DESCRIBE` query will return:

```
:a :b :c .
:a owl:sameAs :a .
:a owl:sameAs :d .
:a owl:sameAs :e .
:d owl:sameAs :a .
:d owl:sameAs :d .
:d owl:sameAs :e .
:e owl:sameAs :a .
:e owl:sameAs :d .
:e owl:sameAs :e .
:d :b :c .
:e :b :c .
```

i.e., without the `<P a rdf:Property>` and `<P rdfs:subPropertyOf P>` statements.

Example 3

If you start with `owl-horst-optimized` and set the `disable-sameAs` parameter to `true` or use `FROM onto:disable-sameAs`, you will receive:

```
:a :b :c .
:a owl:sameAs :d .
:b a rdf:Property .
:b rdfs:subPropertyOf :b .
:d owl:sameAs :e .
```

i.e., the explicit statements + `<type Property>`.

Example 4

This query:

```
PREFIX test: <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

DESCRIBE test:a test:b test:c test:d test:e
FROM onto:implicit
FROM onto:disable-sameAs
```

yields:

```
test:b a rdf:Property .
test:b rdfs:subPropertyOf test:b .
```

because all `owl:sameAs` statements and the statements generated from them (`<:d :b :c>`, `<:e :b :c>`) will not be shown.

Note: The same is achieved with the `disable-sameAs` repository parameter set to `true`. However, if you start with the `empty` ruleset and then switch to a non-empty ruleset, the latter query will not return any results. If you start with owl-horst-optimized and then switch to `empty`, `<type Property>` will persist, i.e., the latter query will return some results.

Example 5

If you use named graphs, the results will look differently:

```
PREFIX test: <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  GRAPH test:graph {
    test:a test:b test:c .
    test:a owl:sameAs test:d .
    test:d owl:sameAs test:e .
  }
}
```

Then the test query will be:

```
PREFIX test: <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

SELECT DISTINCT *
{
  GRAPH ?g {
    ?s ?p ?o
    FILTER (
      ?s IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?p IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?o IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?g IN (test:a, test:b, test:c, test:d, test:e, test:graph)
    )
  }
}
```

If you have started with `owl-horst-optimized`, `disable-sameAs=false`, you will receive:

```
graph {
  :a :b :c .
  :a owl:sameAs :d .
  :d owl:sameAs :e .
}
```

because the statements from the default graph are not automatically included. This is the same as in the `DESCRIBE` query, where using both `FROM onto:explicit` and `FROM onto:implicit` nullifies them.

So, if you want to see all the statements, you should write:

```
PREFIX test: <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

SELECT DISTINCT *
FROM NAMED onto:explicit
FROM NAMED onto:implicit
{
  GRAPH ?g {
    ?s ?p ?o
    FILTER (
      ?s IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?p IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?o IN (test:a, test:b, test:c, test:d, test:e, test:graph) ||
      ?g IN (test:a, test:b, test:c, test:d, test:e, test:graph)
    )
  }
}
ORDER BY ?g ?s
```

Note that when querying quads, you should use the `FROM NAMED` clause and when querying triples — `FROM`. Using `FROM NAMED` with triples and `FROM` with quads has no effect and the query will return the following:

```
:graph {
  :a :b :c .
  :a owl:sameAs :d .
  :d owl:sameAs :e .
}
onto:implicit {
  :b a rdf:Property .
  :b rdfs:subPropertyOf :b .
}
onto:implicit {
  :a owl:sameAs :a .
  :a owl:sameAs :d .
  :a owl:sameAs :e .
  :d owl:sameAs :a .
  :d owl:sameAs :d .
  :d owl:sameAs :e .
  :e owl:sameAs :a .
  :e owl:sameAs :d .
  :e owl:sameAs :e .
}
onto:implicit {
  :d :b :c .
  :e :b :c .
}
```

In this case, the explicit statements `<:a owl:sameAs :d>` and `<:d owl:sameAs :e>` appear also as implicit. They do not appear twice when dealing with triples because the iterators return unique triples. When dealing with quads, however, you can see all statements.

Here, you have the same effects with `FROM NAMED onto:explicit`, `FROM NAMED onto:implicit` and `FROM NAMED`

`onto:disable-sameAs`, and the behavior of the `<type Property>`.

6.7.3.4 RDFS and OWL support optimizations

There are several features in the RDFS and OWL specifications that lead to inefficient entailment rules and axioms, which can have a significant impact on the performance of the inferencer. For example:

- The consequence `X rdf:type rdfs:Resource` for each URI node in the RDF graph;
- The system should be able to infer that URIs are classes and properties, if they appear in schema-defining statements such as `Xrdfs:subClassOf Y` and `X rdfs:subPropertyOf Y`;
- The individual equality property in OWL is reflexive, i.e., the statement `X owl:sameAs X` holds for every OWL individual;
- All OWL classes are subclasses of `owl:Thing` and for all individuals `X rdf:type owl:Thing` should hold;
- `C` is inferred as being `rdfs:Class` whenever an instance of the class is defined: `I rdf:type C`.

Although the above inferences are important for formal semantics completeness, users rarely execute queries that seek such statements. Moreover, these inferences generate so many inferred statements that performance and scalability can be significantly degraded.

For this reason, optimized versions of the standard rulesets are provided. These have `-optimized` appended to the ruleset name, e.g., `owl-horst-optimized`.

The following optimizations are enacted in GraphDB:

Optimization	Affects patterns
Remove axiomatic triples	• `<any> <any> <rdfs:Resource>` • `<rdfs:Resource> <any> <any>` • `<any> <rdfs:domain> <rdf:Property>` • `<any> <rdfs:range> <rdf:Property>` • `<owl:sameAs> <rdf:type> <owl:SymmetricProperty>` • `<owl:sameAs> <rdf:type> <owl:TransitiveProperty>`
Remove rule conclusions	• `<any> <any> <rdfs:Resource>`
Remove rule constraints	• `[Constraint <variable> != <rdfs:Resource>]`

6.8 Administration via HTTP with curl

This page describes GraphDB REST API calls for system administration as cURL commands, which lets developers script these calls in their applications.

See also the *Help ▸ REST API* view of the GraphDB Workbench where you will find a complete reference of all REST APIs and be able to run API calls directly from the browser.

In addition to this, the RDF4J API is also available.

6.8.1 Cluster group management

Get cluster config

```
GET /rest/cluster/config
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/cluster/config'
```

Get cluster group status

```
GET /rest/cluster/group/status
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/cluster/group/status'
```

Get cluster node status

```
GET /rest/cluster/node/status
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/cluster/node/status'
```

Create cluster group

```
POST /rest/cluster/config
```

Example:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
   "electionMinTimeout": 7000,
   "electionRangeTimeout": 5000,
   "heartbeatInterval": 2000,
   "messageSizeKB": 64,
   "nodes": [
     "string"
   ],
   "verificationTimeout": 300
 }' '<base_url>/rest/cluster/config'
```

Add cluster node

```
POST /rest/cluster/config/node
```

Example:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
   "nodes": [
     "string"
   ]
 }' '<base_url>/rest/cluster/config/node'
```

Update cluster group properties

```
PATCH /rest/cluster/config
```

Example:

```
curl -X PATCH --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
   "electionMinTimeout": 7000,
   "electionRangeTimeout": 5000,
   "heartbeatInterval": 2000,
   "messageSizeKB": 64,
   "verificationTimeout": 300
 }' '<base_url>/rest/cluster/config'
```

Delete cluster group

```
DELETE /rest/cluster/config
```

Example:

```
curl -X DELETE --header 'Accept: application/json' '<base_url>/rest/cluster/config?force=false'
```

Delete cluster node

```
DELETE /rest/cluster/config/node
```

Example:

```
curl -X DELETE --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
   "nodes": [
     "string"
   ]
 }' '<base_url>/rest/cluster/config/node'
```

Replace cluster nodes

```
PATCH /rest/cluster/config/node
```

Example:

```
curl -X PATCH --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
   "addNodes": [
     "string"
   ],
   "removeNodes": [
     "string"
   ]
 }' '<base_url>/rest/cluster/config/node'
```

6.8.2 Cluster monitoring

Get cluster statistics

```
GET /rest/monitor/cluster
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/monitor/cluster'
```

6.8.3 Infrastructure monitoring

Get all infrastructure statistics

```
GET /rest/monitor/infrastructure
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/monitor/infrastructure'
```

6.8.4 Location management

Most location management queries can either take the following set of attributes as an argument or return them as a response.

- `active` (boolean): `True` if the location is the currently active one — the local location is the only location that can be active at any given point.
- `defaultRepository` (string): Default repository for the location.
- `errorMsg` (string): Error message, if there was an error connecting to this location.
- `label` (string): Human readable label
- `local` (boolean): `True` if the location is local (on the same machine as the Workbench).
- `password` (string): Password for the new location if any. This parameter only makes sense for remote locations.
- `system` (boolean): `True` if the location is the system location.
- `uri` (string): The GraphDB location URL.
- `username` (string): Username for the new location if any. This parameter only makes sense for remote locations.

Get all connected GraphDB locations

```
GET /rest/locations
```

Example:

```
curl <base_url>/rest/locations
```

Modify settings for a connected GraphDB location

```
POST /rest/locations
```

Example:

```
curl -X POST <base_url>/rest/locations -H 'Content-Type: application/json' -d '
    {
        "username": "<username>",
        "password": "<password>",
        "uri": "<location_uri>"
    }'
```

Connect to a remote GraphDB location

```
PUT /rest/locations
```

Example:

```
curl -X PUT <base_url>/rest/locations -H 'Content-Type: application/json' -d '
    {
        "username": "<username>",
        "password": "<password>",
        "uri": "<location_uri>"
    }'
```

Disconnect a GraphDB location

```
DELETE /rest/locations
```

Example:

```
curl -X DELETE <base_url>/rest/locations?uri=<encoded_location_uri>
```

Set the default repository

```
POST /rest/locations/active/default-repository
```

Example:

```
curl -X POST <base_url>/rest/locations/active/default-repository -H 'Content-Type: application/json' -d
↪'
{
    "repository": "<repo_id>"
}'
```

6.8.5 Repository SHACL validation

You can use the endpoints outlined below to trigger bulk validation on the repository.

Validate data from the repository, identified by `repositoryID` with the shapes stored in a repository, identified by `shapesRepositoryID`:

```
POST: /rest/repositories/{repositoryID}/validate/repository/{shapesRepositoryID}
```

Example:

```
curl -X POST http://localhost:7200/rest/repositories/data-repo/validate/repository/shacl-reco
```

Validate data from the repository, identified by `repositoryID`, with the shapes you send in the request:

```
POST: /rest/repositories/{repositoryID}/validate/text
```

You can send the shapes as a part of the cURL command the following way:

Example:

```
curl -X POST -H "Content-Type: text/turtle"\
    --data-raw "@prefix ex: <http://example.org/ns#> .
                @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
                @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
                @prefix schema: <http://schema.org/> .
                @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
                ex:Aerith a schema:Person ;
                          schema:age \"Two\" ."\
    http://localhost:7200/rest/repositories/data-repo/validate/text
```

Alternatively, you can send it as a file instead:

Example:

```
curl -X POST -H "Content-Type: text/turtle"\
    --data-binary "@data.ttl"\
    http://localhost:7200/rest/repositories/data-repo/validate/text
```

6.8.6 Saved queries

Get saved query (or queries, if no parameter specified)

```
GET /rest/sparql/saved-queries
```

Example:

```
curl <base_url>/rest/sparql/saved-queries?name=<query_name>
```

Create a new saved query

```
POST /rest/sparql/saved-queries
```

Example:

```
curl -X POST <base_url>/rest/sparql/saved-queries -H 'Content-Type: application/json' -d '
{
    "body": "<query_body>",
    "name": "<query_name>"
}'
```

Edit an existing saved query

```
PUT /rest/sparql/saved-queries
```

Example:

```
curl -X PUT <base_url>/rest/sparql/saved-queries -H 'Content-Type: application/json' -d '
{
    "body": "<query_body>",
    "name": "<query_name>"
}'
```

Delete an existing saved query

```
DELETE /rest/sparql/saved-queries
```

Example:

```
curl -X DELETE <base_url>/rest/sparql/saved-queries?name=<query_name>
```

6.8.7 Security management

Check if security is enabled

```
GET /rest/security
```

Example:

```
curl <base_url>/rest/security
```

Enable security

```
POST /rest/security
```

Example:

```
curl -X POST --header 'Content-Type: application/json' -d true <base_url>/rest/security
```

Check if free access is enabled

```
GET /rest/security/free-access
```

```
curl <base_url>/rest/security/free-access
```

Enable or disable free access

```
POST /rest/security/free-access
```

Example:

```
curl -X POST --header 'Content-Type: application/json' -d '
    {
        "appSettings": {
              "DEFAULT_SAMEAS": boolean,
              "DEFAULT_INFERENCE": boolean,
              "EXECUTE_COUNT": boolean,
              "IGNORE_SHARED_QUERIES": boolean
              },
        "authorities": ["string"],
        "enabled": boolean
    }'  '<base_url>/rest/security/free-access'
```

Get all users

```
GET /rest/security/users
```

Example:

```
curl <base_url>/rest/security/users
```

Get a user

```
GET /rest/security/users/<username>
```

Example:

```
curl <base_url>/rest/security/users/<username>
```

Delete a user

```
DELETE /rest/security/users/<username>
```

Example:

```
curl -X DELETE <base_url>/rest/security/users/<username>
```

Change settings for a user

```
PATCH /rest/security/users/<username>
```

Example:

```
curl -X PATCH --header 'Content-Type: application/json' -d '
    {
        "appSettings": {
              "DEFAULT_SAMEAS": boolean,
              "DEFAULT_INFERENCE": boolean,
              "EXECUTE_COUNT": boolean,
              "IGNORE_SHARED_QUERIES": boolean
              },
        "password": "string"
   }'   '<base_url>/rest/security/users/<username>'
```

Create a user

```
POST /rest/security/users/<username>
```

Example:

```
curl -X POST --header 'Content-Type: application/json' -d '
    {
        "appSettings": {
              "DEFAULT_SAMEAS": boolean,
              "DEFAULT_INFERENCE": boolean,
              "EXECUTE_COUNT": boolean,
              "IGNORE_SHARED_QUERIES": boolean
              },
        "grantedAuthorities": ["string"],
        "username": "string",
        "password": "string"
    }'  '<base_url>/rest/security/users/<username>'
```

Edit a user

```
PUT /rest/security/users/<username>
```

Example:

```
curl -X PUT --header 'Content-Type: application/json' -d '
    {
        "appSettings": {
              "DEFAULT_SAMEAS": boolean,
              "DEFAULT_INFERENCE": boolean,
              "EXECUTE_COUNT": boolean,
              "IGNORE_SHARED_QUERIES": boolean
              },
        "grantedAuthorities": ["string"],
        "username": "string",
        "password": "string"
    }'  '<base_url>/rest/security/users/<username>'
```

6.8.8 SPARQL template management

Create, edit, delete, and execute SPARQL templates, as well as to view all templates and their configuration.

Get IDs of all configured SPARQL templates per current repository

```
GET /rest/repositories/<repo_id>/sparql-templates
```

Example:

```
curl '<base_url>/rest/repositories/<repo_id>/sparql-templates'
```

Get a SPARQL template configuration

```
GET /rest/repositories/<repo_id>/sparql-templates/configuration
```

Example:

```
curl -X GET --header 'Accept: text/plain' '<base_url>/rest/repositories/<repo_id>/sparql-templates/
↪configuration?templateID=<template_id>'
```

Create a new SPARQL template

```
POST /rest/repositories/<repo_id>/sparql-templates
```

Example:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: */*' -d '{
    "query": "<update_query_string> }",
    "templateID": "<template_id>"
}' '<base_url>/rest/repositories/<repo_id>/sparql-templates'
```

Delete an existing SPARQL template

```
DELETE /rest/repositories/<repo_id>/sparql-templates
```

Example:

```
curl -X DELETE --header 'Accept: */*' '<base_url>/rest/repositories/<repo_id>/sparql-templates?
↪templateID=<template_id>'
```

Edit an existing SPARQL template

```
PUT /rest/repositories/<repo_id>/sparql-templates
```

Example:

```
curl -X PUT --header 'Content-Type: text/plain' --header 'Accept: */*' -d '<update query_string> ' '
↪<base_url>/rest/repositories/<repo_id>/sparql-templates?templateID=<template_id>'
```

Execute an existing SPARQL template

```
POST /rest/repositories/<repo_id>/sparql-templates/execute
```

Example (based on *this data*):

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
     "sugar" : "none" ,
     "year" : 2020 ,
     "s" : "http://www.example.com/demo/wine#Blanquito"
 }' '<base_url>/rest/repositories/<repo_id>/sparql-templates/execute'
```

6.8.9 SQL views management

Access, create, and edit SQL views (tables), as well as delete existing saved queries and see all SQL views for the active repository.

Get all SQL view names for current repository

```
GET /rest/sql-views/tables
```

Example:

```
curl -X GET --header 'Accept: application/json' --header 'X-GraphDB-Repository: <repoID>' '<base_url>/
↪rest/sql-views/tables'
```

Get a SQL view configuration

```
GET /rest/sql-views/tables/<name>
```

Example:

```
curl -X GET --header 'Accept: application/json' --header 'X-GraphDB-Repository: <repoID>' '<base_url>/
↪rest/sql-views/tables/<name>'
```

Create a new SQL view

```
POST /rest/sql-views/tables/
```

Example:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: */*' --header 'X-GraphDB-
↪Repository:<repoID> -d '{
    "name": "string",
    "query": "string",
    "columns": ["string"]
    }'
    <base_url>/rest/sql-views/tables/
```

Edit an existing SQL view

```
PUT /rest/sql-views/tables/<name>
```

Example:

```
curl -X PUT --header 'Content-Type: application/json' --header 'Accept: */*' --header 'X-GraphDB-
↪Repository:<repoID>' -d '{
    "name": "string",
    "query": "string",
    "columns": ["string"]
    }'
    <base_url>/rest/sql-views/tables/<name>
```

Delete an existing saved query

```
DELETE /rest/sql-views/tables/<name>
```

Example:

```
curl -X DELETE <base_url>/rest/sql-views/tables/<name>
```

6.8.10 Authentication

Obtain a GDB token in exchange for username and password

```
POST /rest/login/**
```

Example:

```
curl <base_url>/rest/login/<username> -X POST -H 'X-GraphDB-Password: <password>'
```

This command will return the user's roles and GraphDB applications settings. It will also generate a GDB token, which is returned in an Authorization header and will be used at every next authentication request.

6.8.11 Structures monitoring

Get structures statistics

```
GET /rest/monitor/structures
```

Example:

```
curl -X GET --header 'Accept: application/json' '<base_url>/rest/monitor/structures'
```

6.9 Migrating GraphDB configurations

To migrate from one GraphDB version to another, follow the instructions in the last column of the table below, and then the steps described further down in this page.

6.9.1 Compatibility between the versions of GraphDB, Connectors, and third-party connectors

GraphDB	RDF4J	Connectors	Elasticsearch	OpenSearch	Lucene	Solr	Kafka
11.3.x	5.2.2	16.6.0	9.1.3	3.1.0	9.12.0	9.4.1	3.9.1

Continued on next page

Table 1 – continued from previous page

GraphDB	RDF4J	Connectors	Elasticsearch	OpenSearch	Lucene	Solr	Kafka

	Starting with GraphDB 11.3, the default transport mode used by the MCP server is now streamable HTTP transport, in accordance with the changes to the MCP protocol. Communication through the transport is now done through `HTTP POST` and `GET` requests. SSE transport, which was the default mode prior to GraphDB 11.3, is still supported, and can be enabled through the `graphdb.mcp.transport.mode` configuration property. For more information, check the *Using GraphDB's LLM tools with MCP clients* documentation, as well as the list of *MCP server properties*.
	GraphDB 11.3 fixes an issue with the `op:subtract-dates` operations. Prior to GraphDB 11.3, when calculating the substraction of two dates, GraphDB returned type of `xs:duration`, which is incompliant with the W3C specification on XPath and XQuery Functions and Operators. These operations should now return `xs:dayTimeDuration` as prescribed by the SPARQL specification.
	The following endpoints in `RecoveryController` which were deprecated in GraphDB 10.8 have now been removed:
	• `@GetMapping(value = "/backups")`
	• `@GetMapping(value = "/backups/{name}")`
	• `@PostMapping(value = "/backups/{name}")`
	• `@PutMapping(value = "/backups/{name}")`
	• `@DeleteMapping(value = "/backups/{name}")`
	• `@PostMapping(value = "/cloud-restore", consumes = "application/json")`
	• `@PostMapping(value = "/cloud-backup", consumes = "application/json")`
	• `@PostMapping(value = "/restore/{name}")`
	Starting with version 10.8, the backupOptions object is no longer supported. Users must move its properties to the top level of the request payload. Before 10.8:
	`{ "backupOptions": { "backupSystemData": true }, "bucketUri": "s3:///<bucket-name>/<backup-name>?region=<region>&ACCESS_KEY_ID=<key-id>&_ACCESS_KEY=<key>" }`
	From 10.8 onward:
	`{ "backupSystemData": true, "bucketUri": "s3:///<bucket-name>/<backup-name>?region=<region>&ACCESS_KEY_ID=<key-id>&_ACCESS_KEY=<key>" }`
	Users should update their requests accordingly, as the backupOptions wrapper is no longer recognized.
	The default value for `graphdb.ttyg.model.context.window.size` in GraphDB 11.3 has been changed from `128,000` to `1,000,000`.						

11.2.x	5.1.6	16.5.0	9.1.13	3.1.0	9.12.0	9.4.1	3.9.1

Continued on next page

Table 1 – continued from previous page

GraphDB	RDF4J	Connectors	Elasticsearch	OpenSearch	Lucene	Solr	Kafka
	GraphDB 11.1 fixes an issue where SHACL constraints might not be replicated across the cluster, leading to inconsistencies or missing constraint enforcement on follower nodes. Since the default RDF4J SHACL graph cannot be queried using SPARQL, you can verify whether this problem affects you navigatin to the Graphs overview view and selecting the default RDF4J SHACL graph. If there was a leader change and new shapes were added the shapes graph could be filled with different shapes. Before upgrading to GraphDB 11.2, to ensure all SHACL constraints are replicated and active on all nodes, you need to follow the steps below: 1. Before upgrading, delete your cluster in the previous version of GraphDB 2. For each of the nodes, export the content of the default SHACL graph in that node 3. Delete the default SHACL graph on all of the nodes 4. Upgrade the instances to GraphDB 11.2 version 5. Recreate the cluster on GraphDB 11.2 6. Import all exported SHACL graphs back into the cluster checking the Preserve bNode Ids This will guarantee that the SHACL shapes are now replicated across all nodes. If any duplicates exist, no issues occur, as the system is idempotent. The refresh flag is automatically triggered, so preloaded shapes are active. The names of two properties that configure the truststore were changed in GraphDB 11.2: • `graphdb.connector.truststoreFile` is now `graphdb.connector.ssl.truststoreFile` • `graphdb.conector.truststorePass` is now `graphdb.connector.ssl.trustorePassword` If your deployment is using either of these properties, you need to update your configuration file to reflect this change and use the new property names. GraphDB 11.2 fixes an issue where X.509 client certificate authentication was not functioning correctly from version 11.0. This means that users relying on X.509 based authentication may experience failed client authentication after upgrading from GraphDB 10 to 11. To avoid this issue, users upgrading from version 10 who depend on X.509 support must upgrade directly to GraphDB 11.2 or later, where the functionality is restored. As of RDF4J 5.1.6, the SPARQL TSV writer adheres more strictly to the official TSV specification. Unlike CSV, the TSV format requires IRIs, blank nodes, and literals to be encoded following Turtle rules. Previously, RDF4J produced unquoted xsd:string literals, but from 5.1.6 onward all literals are output using quotation marks. This change may affect users who rely on the old unquoted literal format. **Warning:** Due to a known bug in GraphDB 11.2 and 11.3, a cluster cannot be created if the properties `graphdb.connector.ssl.truststoreFile` and `graphdb.connector.ssl.truststorePassword` are configured through the `graphdb.properties` file. This behavior is not intentional and will be fixed in GraphDB 11.3.3. You can still create a cluster if you pass those configuration properties as JAVA arguments: `-Djavax.net.ssl.trustStore=<path-to-truststore> -Djavax.net.ssl.trustStorePassword=<password>` instead of having them in the `graphdb.properties` file. We recommend using this workaround until this bug is fixed.						
11.1.x	5.1.4	16.4.0	9.0.2	3.1.0	9.12.0	9.4.1	3.9.1

Continued on next page

Table 1 – continued from previous page

GraphDB	RDF4J	Connectors	Elasticsearch	OpenSearch	Lucene	Solr	Kafka
	GraphDB 11.1 enhances the *Talk to Your Graph* feature and other mechanisms that use Large Language Models (LLMs) to support alternatives to OpenAI LLMs. As a result, several configuration properties related to those mechanisms have been deprecated and replaced with new properties: • `graphdb.llm.api` is a new property that configures the LLM API that will be used by GraphDB. • `graphdb.openai.api-key` has been replaced by `graphdb.llm.api-key`. • `graphdb.openai.url` has been replaced by `graphdb.llm.url`. • `graphdb.openai.auth` has been replaced by `graphdb.llm.auth`. • `graphdb.llm.model` is a new property that configures the default LLM model for TTYG and GPT functions and SPARQL explain. While we strongly suggest updating your configuration properties to reflect those changes, the deprecated properties are still supported. For more information, check the configuration properties documentation section in *Talk to Your Graph and LLM properties*. GraphDB no longer supports non-global cache. As a result, `graphdb.global.page.cache` has been deprecated and has no effect. The `verificationTimeout` *configuration property used by the cluster mechanism* has a new default value of `300` (it was previously `1500`). This does not affect already existing cluster configurations. GraphDB 11.1.1 fixes a bug which caused a fingerprint mismatch when using connectors. Since the issue can leave invalid data in `dynamic_settings.json`, it is advised to recreate and reindex the connector instances to the indexing of unrelated documents.						
11.0.x	5.1.2	16.3.1	8.14.3	2.18.0	9.12.0	9.4.1	3.9.0

Continued on next page

Table 1 – continued from previous page

GraphDB	RDF4J	Connectors	Elasticsearch	OpenSearch	Lucene	Solr	Kafka

	GraphDB 11 comes with a higher Java version requirement (Java 21). You need to upgrade your version of Java to Java 21.
	GraphQL support is now part of GraphDB 11. It was previously supported in Semantic Objects. If you want to migrate from Semantic Objects to GraphDB 11, check the *Migrating GraphQL configurations from Semantic Objects to GraphDB* documentation.
	GraphDB 11 now uses Logback 1.5, which affects the format of the `logback.xml` file:
	• In configuration files, `<if>` elements can no longer be nested within an `<appender>`, `<logger>` or `<root>` elements.
	• `<jmxConfigurator />` was removed from Logback and is no longer supported in GraphDB.
	Some GraphDB configuration properties have been deprecated and replaced with new ones:
	• `default.min.distinct.threshold` has been deprecated and replaced by `graphdb.query.memory.threshold`.
	• Due to the migration to Tomcat 11.0.5, some network configuration properties have been renamed. Check the *Network properties* documentation for full information on those properties.
	• SSL can now be enabled with a single property `graphdb.connector.ssl.enabled`. Setting this property to `true` is equal to this configuration used in versions prior to GraphDB 11:
	`graphdb.connector.SSLEnabled = true`
	`graphdb.connector.scheme = https`
	`graphdb.connector.secure = true`
	Starting with GraphDB 11, subtree search is available and enabled by default for both users and roles. Previously, subtree search was available by default only for users, without an option to control it. For more information, check the *LDAP user database* documentation.
	The following licensing and feature changes affect GraphDB Free:
	• **License requirement**: The built-in license for GraphDB Free has been removed. Users upgrading to GraphDB Free 11 must request a free license from the Ontotext website and manually install it before use.
	• **Repository limit**: GraphDB Free now supports up to 5 repositories.
	• **Authentication and Authorization**: LDAP, OpenID/OAuth, Kerberos, and X.509 authentication and authorization methods are no longer available in GraphDB Free.
	• **Entity ID restriction**: Support for 40-bit entity IDs has been removed.
	All these features remain available in GraphDB Enterprise and GraphDB Standard.
	If you are upgrading from GraphDB 10.x or older, please upgrade to GraphDB 10.8 before you upgrade to GraphDB 11.0.						

See compatibility for GraphDB versions 10.x and older.							

6.9.2 Migrating GraphDB

To migrate your GraphDB configuration and data, follow the steps below.

> **Warning:** Keep in mind that after that, you cannot automatically revert to GraphDB 10.x.

1. Stop the GraphDB 10.x instance.
2. Back up your repositories and configuration — this will ensure you can revert safely if something goes wrong during the upgrade.
   a. To back up all repositories, copy the `data` directory. See also Backing up and Restoring a Repository for additional ways to backup repository data.
   b. To back up all configuration, copy the `work` and `conf` directories.
3. Your existing GraphDB 10.x home directory (containing the `conf`, `data`, and `work` directories) can be used directly as the GraphDB 11.0 home directory.

> **Tip:** You can also copy the `conf`, `data`, and `work` directories from the GraphDB 10.x home directory to a new directory to use as the GraphDB 11.0 home directory. In this case, your GraphDB 10.x home directory is also the backup so you may skip the backup steps.
>
> The various directories are described in detail in the *Structure of the distribution package* section.

4. Start GraphDB 11.0.

6.9.3 Migrating plugins in a cluster

When upgrading to a newer GraphDB version, it might contain plugins that are not present in the older version. In this case, and when using a cluster, the Plugin Manager disables the newly detected plugins, so you need to enable them by executing the following SPARQL query:

```
insert data {
    [] <http://www.ontotext.com/owlim/system#startplugin> "plugin-name"
}
```

Then create your plugin following the steps described in the corresponding documentation, and also make sure to not delete the database in the plugin you are using.

You can also stop a plugin before the migration in case you deem it necessary:

```
insert data {
    [] <http://www.ontotext.com/owlim/system#stopplugin> "plugin_name"
}
```

6.10 Migrating GraphQL configurations from Semantic Objects to GraphDB

Previous to GraphDB 11, GraphQL integration with GraphDB was supported throuhg Semantic Objects. If you have previously used GraphQL in Semantic Objects and need to migrate from Semantic Objects to GraphDB, there are some important differences you need to keep in mind.

6.10.1 General notes

- The SHACL validation functionality has not been migrated in GraphDB 11.0 and is planned for a later date.
- GraalVM based IRI and value generation has been fully migrated to Kotlin based implementation. However, some custom expressions may need to be updated to be compatible with the Kotlin based expressions format. Note that custom Kotlin expressions may be slower than the GraalVM JS expressions.
- The Schema Validation against data has been partially migrated. The persistent validation logic is not available in GraphDB 11, and only the immediate in-memory version is present.

6.10.2 Licence changes

GraphQL is generally available in every version of GraphDB 11. The following additional features require a GraphDB GraphQL extended license.

- Role-based access control (RBAC)
- SPARQL federation
- Apollo Federation
- Schema validation against data

Warning: Existing Semantic Objects licenses cannot be directly used in GraphDB 11. If you already have a Semantic Objects license, contact your representative or Graphwise sales to receive an updated license.

6.10.3 Deployment migration

1. Do not deploy Semantic Objects and Semantic Objects Workbench.
2. Deploy GraphDB 11.
3. Import the models into your GraphDB repositories.

6.10.4 API Calls

The GraphQL API available in Semantic Objects has been integrated in GraphDB 11.0 with the following changes:

- The GraphQL API has been changed to be per repository as well as to have multiple schemas per repository.
- The old Semantic Objects GraphQL endpoint `/graphql` has been changed to `/repositories/<repository>/graphql` for the default GraphQL schema.
- There are also endpoints per repository that are `/rest/repositories/<repository>/graphql/<endpoint_id>`.

Note: Semantic Objects 5.0 also has the `/graphql/<model_id>` that matches the GraphDB multischema definition.

The following controllers have been removed:

- `POST /soml/{id}/validate`
- `GET /soml/{id}/validate`
- `GET /soml/{id}/validate/{taskId}`

The following controllers have been replaced:

Old controller	New controller
`GET /soml`	`GET /rest/repositories/{repositoryId}/graphql/manage/endpoints`

`GET /soml/{id}`	`GET /rest/repositories/{repositoryId}/graphql/manage/endpoints/{id}`
	Or:
	`GET /rest/repositories/{repositoryId}/graphql/manage/endpoints/{id}/export`

`POST /soml`	`POST /rest/repositories/{repositoryId}/graphql/manage/endpoints/import`
`PUT /soml/{id}`	`POST /rest/repositories/{repositoryId}/graphql/manage/endpoints/import`
`POST /soml/validate`	`POST /rest/repositories/graphql/manage/validate`
`GET /soml/{id}/validateNow`	`GET /rest/repositories/{repository}/graphql/manage/endpoints/{endpointId}/validate`

`PUT /soml/{id}/soaas`	`POST /rest/repositories/{targetRepo}/graphql/manage/endpoints/{endpointId}`
	with the following body:
	`{ "default": true }`

6.10.5 Configuration parameters

All configurations have been moved to a new context and now begin with a `graphdb.` prefix. For more information on changes to the configuration parameters, check the *Directories and configuration properties* documentation.

6.10.6 Schema changes

There is now an option to configure the federated service via the Semantic Objects Modelling Language model (SOML model) using the following configuration:

```
config:
    sparqlFederatedServices.<service>: <service_url>
    sparqlFederatedServicesPriority: CONFIG_FIRST|MODEL_FIRST
```

Where the options for `sparqlFederatedServicesPriority` are:

- `CONFIG_FIRST`: Prioritize the deployment config over the model configurations. Prevents changes in the models to affect the deployment. Mainly applicable for dynamic external services.
- `MODEL_FIRST`: Prioritize the model configurations over deployment configurations. Allows for transferring the models without the need to modify the deployment options. Mainly usable for static external services.

6.10.6.1 Schema migration

There are no breaking changes to the SOML that require special care when migrating schemas from Semantic Objects 5.0 to GraphDB 11.

Migration of SOML from SO to GraphDB

Migration via file copy

1. Download the SOML file from Semantic Objects using the following endpoint:

```
GET /soml
```

2. Import the SOML into the relevant repository in GraphDB where GraphQL will be used.
3. Optionally, set the schema as default so that it's accessible at the default repository endpoint.

Migration via system repository export

1. Open the old or migrated GraphDB that was previously connected to Semantic Objects.
2. Export the `otp-system` repository.
3. Import the data into the relevant repository in GraphDB where GraphQL will be used.
4. Execute the following SPARQL update that corrects the default schema:

```
prefix so: <http://www.ontotext.com/semantic-object#>

delete {
    ?s so:bound-schema ?o
}
insert {
    graph ?g {
        so:defaultSchemaId so:bound-schema ?o .
    }
}
where {
    graph ?g {
        ?s so:bound-schema ?o.
    }
}
```

> **Note:** This should update only a single row in the data.

Migration where the whole system repository cannot be exported

1. Open the old or migrated GraphDB that was previously connected to Semantic Objects.
2. Export the `http://www.ontotext.com/semantic-object#store`.
3. Import the exported data into the relevant repository in GraphDB where GraphQL will be used.
4. Optionally, set the default schema.
5. Optionally, export `http://www.ontotext.com/semantic-object#id-store` and import it into the relevant target repository.
6. Optionally, run the SPARQL update found under **Migration via system repository export** above.

6.10.7 RBAC migration

Role-Based Access Control is fully supported in GraphDB 11 with the following changes:

- The default role will allow all access by default, but this can be redefined.
- Filters with JWT claims are replaced with filters with GraphDB username and GraphDB user roles.
- `ctx.claims.preferred_username` will resolve to the GraphDB user username.
- `ctx.claims.authorities` will resolve to the GraphDB user custom roles without the role `CUSTOM_` prefix.

Tip: It is a good idea to replace `ctx.claims.preferred_username` and `ctx.claims.authorities` with their new alternative names:

- `ctx.claims.preferred_username` ‣ `ctx.username`
- `ctx.claims.authorities` ‣ `ctx.roles`

6.11 How to deploy GraphDB in AWS

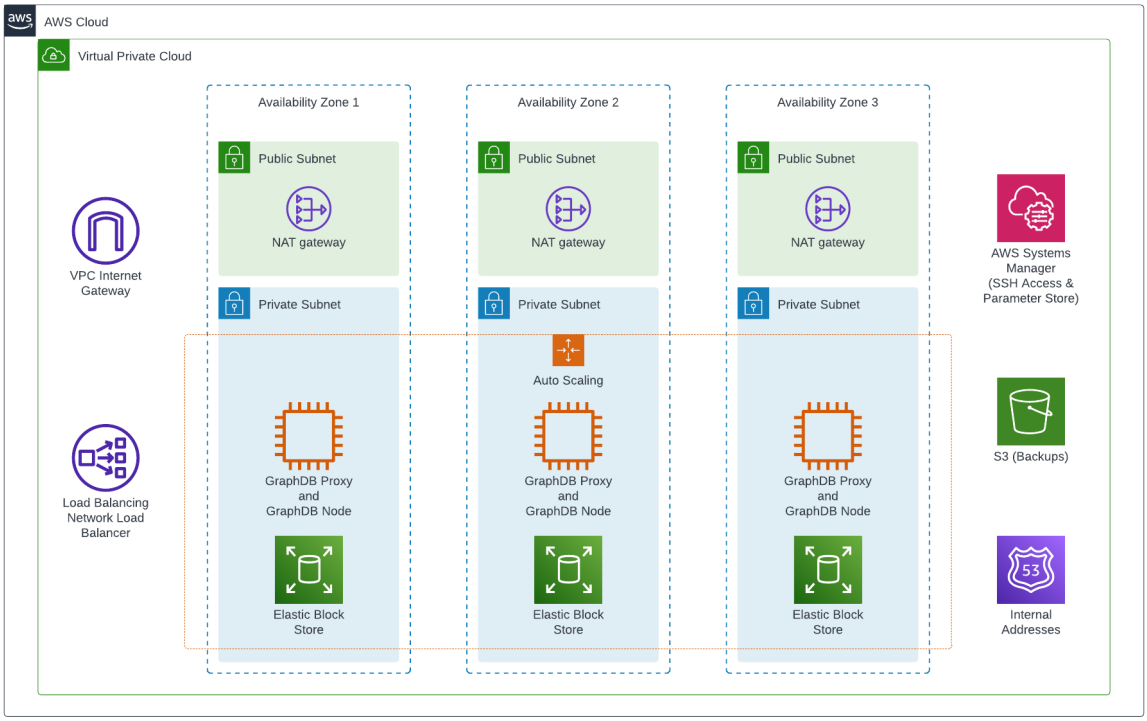
GraphDB can be deployed on Amazon Web Services by following the *general installation instructions*. You can find information regarding the costs of running a GraphDB instance on the AWS Services website.

This documentation will walk you through the process of setting up the necessary environment for deploying GraphDB on AWS.

Note: Ontotext maintains a Terraform module that automates this entire procedure. Learn more about how to use it at our GitHub repository.

6.11.1 Architecture

The GraphDB architecture diagram showcases the deployment architecture for GraphDB on EC2 instances in AWS cloud platform. The diagram illustrates the key components, and their interactions to provide a high-level understanding of the system's architecture and how it should be deployed.



Note: There are no third-party integration points on the default GraphDB deployment.

6.11.2 Prerequisites

There are several prerequisites for running a GraphDB instance on AWS:

- Access to an AWS account (we recommend the use of an Identity and Access Management user for the deployment instead of a root user account)
- Active *GraphDB license* required to use the Enterprise functionalities of the database
- Create a shell script used to initialize the EC2 instance

Note: The GraphDB Terraform module contains a Terraform templates you can use when creating your shell script. If you use the Terraform templates, you will need to replace the placeholder values of all variables with your actual values.

6.11.2.1 Required skills

Note: Deploying GraphDB on AWS EC2 requires a combination of skills in AWS infrastructure management, database administration, and system troubleshooting. Acquiring these skills may involve hands-on experience, self-study, online resources, and formal training programs provided by AWS or other educational platforms.

The following skills and knowledge are typically required in order to successfully deploy GraphDB on AWS EC2:

AWS Fundamentals	Familiarity with Amazon Web Services (AWS) and understanding of its core concepts, such as EC2 instances, security groups, VPCs and IAM roles. Knowledge of how to navigate the AWS Management Console and interact with AWS services is essential.
EC2 Instance Management	Proficiency in creating and managing EC2 instances. This includes selecting the appropriate instance type, configuring security settings, managing storage (EBS volumes), and understanding EC2 instance lifecycle management.
Networking and Security	Understanding of networking concepts in AWS, including VPC (Virtual Private Cloud) configuration, subnets, routing tables, and security groups. Knowledge of how to set up inbound and outbound traffic rules to allow communication with GraphDB.
Linux Administration	Proficiency in Linux command-line interface (CLI) and basic administration tasks. This includes SSH access to EC2 instances, navigating the file system, managing permissions, installing packages, and configuring system settings.
Database Management	Knowledge of GraphDB and its deployment requirements. Understanding of how to configure GraphDB settings, including database storage, memory allocation, and repository creation.
Database Backup and Recovery	Familiarity with backup and recovery strategies for GraphDB on AWS. Knowledge of AWS services like Amazon S3 for data backups and restoration processes.
Monitoring and Troubleshooting	Proficiency in monitoring the health and performance of GraphDB instances on AWS. Understanding of logging, monitoring and troubleshooting techniques using AWS CloudWatch, EC2 instance logs, and GraphDB diagnostic tools.
High Availability and Scalability	Knowledge of implementing high availability and scalability for GraphDB on AWS. This may involve using features like EC2 Auto Scaling, load balancers, and multi-Availability Zone (AZ) deployments.
Infrastructure as Code (IaC)	Familiarity with Infrastructure as Code principles and tools like AWS CloudFormation or Terraform. This enables automating the provisioning and configuration of GraphDB infrastructure on AWS.
Security Best Practices	Understanding of security best practices for AWS deployments, including data encryption, access controls, identity and access management, and compliance considerations.

6.11.2.2 AWS billable services required by GraphDB

The table below lists the minimum billable and non-billable AWS services to deploy GraphDB on AWS.

Note: Please refer to the AWS Services website for the latest AWS documentation and information on pricing.

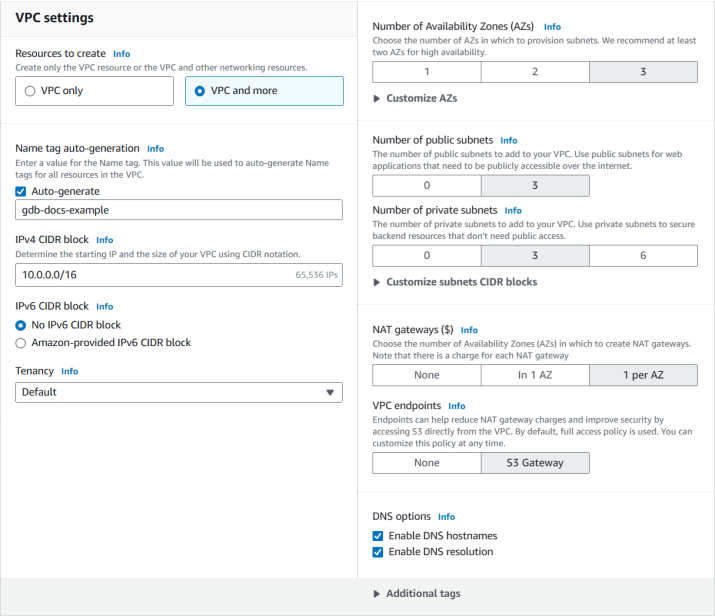
AWS Service	Description
Virtual Private Cloud (VPC)	Allows you to create a logically isolated virtual network for your GraphDB depl
AWS Systems Manager	License and various GraphDB configurations are saved in the Parameter Store
AWS Identity and Access Management (IAM)	Provides user and access management for your GraphDB deployment
Elastic Compute Cloud (EC2)	A server instance that Elastic Kubernetes Service will be using as a managed no

Table 2 – continued from previo

AWS Service	Description
Network Load Balancer (NLB)	Load balancing service that distributes incoming network traffic across GraphD
Elastic Block Store (EBS)	Provides persistent block-level storage volumes for EC2 instances
Simple Storage Service (S3)	Used to store GraphDB data backups
AWS CloudWatch	Enables monitoring and logging to *monitor the status of the GraphDB cluster*
AWS Support	AWS Support for troubleshooting AWS service specific issues

6.11.3 Setting up your Virtual Private Cloud (VPC)

1. Go to the *VPC Management Console* and click on *Create VPC*
2. Select the *VPC and more* option. This will allow you to configure and create all other networking components such as subnets, gateways, and more
3. Enter the following VPC configurations:
   1. *Name tag auto-generation:* Enter a descriptive name by which to recognize your VPC
   2. *Number of Availability Zones:* 3
   3. *NAT Gateways:* 1 per AZ *(this will enable external Internet access)*
4. Check both the *Enable DNS hostnames* and *Enable DNS resolution* checkboxes



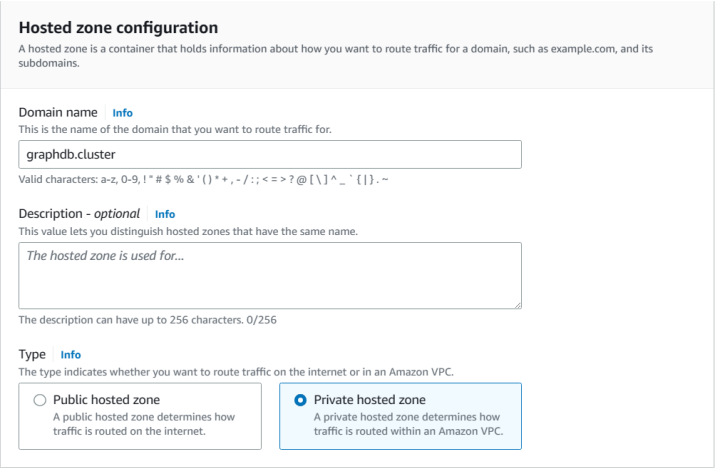
5. Click on *Create VPC*

Once you've completed this process, you will see various status messages as the system creates the subnets, NAT gateways, and other components of the VPC. This may take several minutes; you will know that it is finished when all the status messages have turned green and the *View VPC* button appears at the bottom.

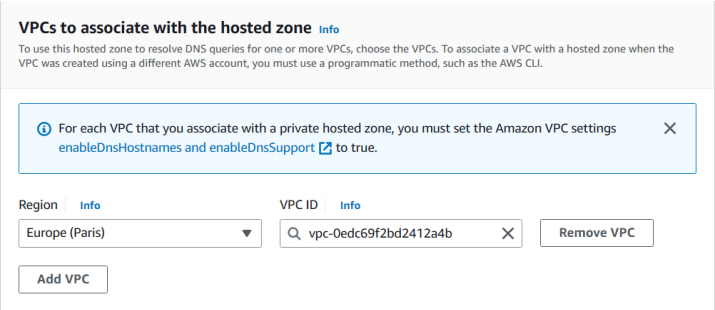
6.11.4 Setting up your Route 53 private hosted zone

The GraphDB Raft implementation requires static addresses. This is achieved by creating a private hosted zone in Route 53 and registering the instances there.

1. Go to the Route 53 dashboard and select *Hosted Zones* from the navigation menu on the left
2. Click *Create hosted zone*
3. Enter a domain name, such as `graphdb.cluster`



4. Choose *Private hosted zone*
5. Under the *VPC settings* select your region and the VPC that you created



6. Click *Create hosted zone*

Tip: You may want to write down the *Hosted Zone ID*, as you will need it later.

Note: *Later,* you will also need to create "A" records for the instances.

6.11.5 Creating an S3 bucket

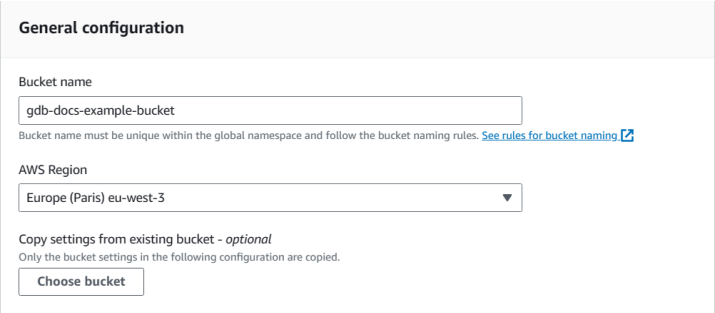
Tip: This step is **optional**, but recommended.

GraphDB can store backups to S3 and, if needed, restore from them.

Note: Backup with GraphDB relies on AWS S3 Transfer Acceleration, which is limited to select AWS regions. See Amazon's documentation on Requirements for using Transfer Acceleration for information on available Regions and other requirements.

To create an S3 bucket:

1. Go to the S3 console and click on *Create bucket*
2. Enter a name for the bucket that is *globally unique among all S3 buckets*



3. Select your region, scroll down and click *Create bucket*

Once you've completed this process, you should then see a *"Successfully created"* message. We recommend you to block all public access, so that it isn't accessible by anyone else.

6.11.6 Importing a certificate into Amazon Certificate Manager (ACM)

Tip: This step is **optional**.

While serving GraphDB requests over a secured and encrypted connection is not strictly required, it is highly recommended. This section goes over the process of importing a certificate into Amazon Certificate Managed (ACM) which could be used in the next section while *creating the load balancer*.

1. Go to the *Certificate Manager* console and select *Import certificate* from the navigation menu on the left, or click on *Import* at the top
2. Paste the *PEM encoded certificate body*
3. Paste the *PEM encoded private key*

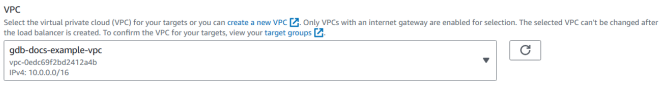
   Warning: You need to remove the password for the key before pasting it

4. You can optionally paste the certificate chain, then click on *Next*
5. You can optionally add tags, then click on *Next* again
6. Click on *Import*

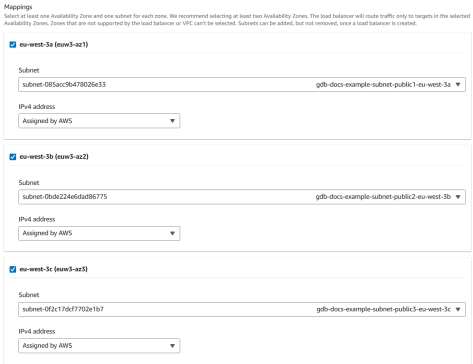
Open the imported certificate and note the **Amazon Resource Name (ARN)** for it — you will need it when creating the load balancer lister.

6.11.7 Setting up the Load Balancer

1. Go to the *EC2 Dashboard* and select *Load Balancers* from the navigation menu on the left
2. In the top-right, make sure you are in the region for which you want to create a Load Balancer
3. Click on *Create load balancer*
4. Click on *Network Load Balancer* and enter the following configurations:
    1. *Load balancer name:* Enter a descriptive name by which to recognize your load balancer
    2. Choose the *Internet-facing scheme (otherwise GraphDB will not be accessible externally)*
    3. *VPC:* choose the VPC that was previously created



    4. *Mappings:* select all three availability zones and the public subnets in them



    5. Under *Security groups*, remove the *Default* security group
5. Optionally, you can create a TLS listener and remove the TCP listener on port 80:
    1. Remove the *TCP listener* on port 80 *(leaving it will result in unencrypted traffic)*
    2. Click on *Add listener* and choose the following configurations:
        1. *Protocol*: select *TLS* from the previous step
        2. *Port*: `443`
    3. *Default action*: select the target group from the previous step
    4. *Secure Listener settings*:
        1. Leave the security policy to the recommended one
        2. Select the certificate that you imported in ACM and leave the ALPN policy set to `None`
6. Click on *Create target group* under Listener and routing
7. Enter the following configurations:
    1. *Target type:* leave as instances
    2. *Target group name:* Enter a descriptive name by which to recognize your group
    3. *Protocol:* TCP
    4. *Port:* 7200
    5. *Health check path:* `/rest/cluster/node/status`
    6. Under *Advanced health check settings*, override the port to be `7201`

7. Go back to the load balancer creation page and select the target group that was created in the previous step

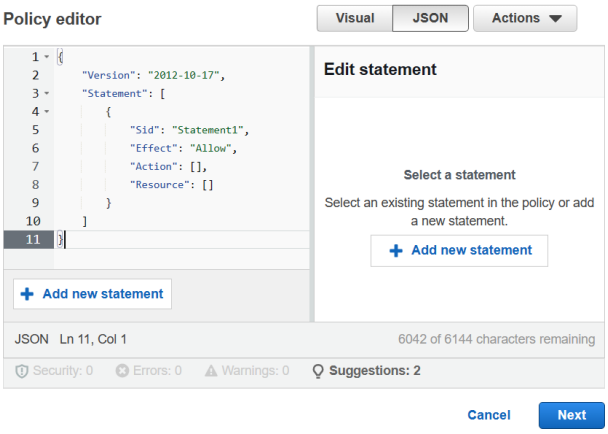
8. Return to the *Load Balancer* page and refresh the list of Target Groups

9. Select the one you just created, and click on *Create load balancer*

Note: The TCP idle connection timeout is 350 seconds. To prevent timeouts, we recommend the inclusion of Keep-Alive messages with your requests.

6.11.8 Setting up instance role and profile permissions

You have to grant GraphDB EC2 instances certain permissions so that they can perform several tasks in AWS. This section describes what permissions they need, what they are used for, and how to create them. To do this, we will create an instance profile and then attach it to the instances.



To create a policy:

1. Go to the *Identity and Access Management* (IAM) dashboard and select *Policies* from the navigation menu on the left

2. Click on *Create policy* and go to the *JSON tab*

3. Replace the JSON script with the one for the permission you are creating

4. Click on *Next*

5. Enter a *Policy name* in the *Policy details section* of the *Review and create* screen

6. Click on *Create policy*

Warning: You need to create a different policy for each of the JSON scripts listed below.

6.11.8.1 Allow the EC2 instance to read, write, and list objects in S3 (optional)

If you are planning to store GraphDB backups to S3, the EC2 instance needs to be able to read, write, and list objects in S3. To do this, paste the following JSON in the appropriate field when creating a policy, changing `graphdb-backup-bucket` to the name of your S3 bucket:

```
{
"Version":"2012-10-17",
"Statement":[
    {
```

(continues on next page)

(continued from previous page)

```
        "Effect":"Allow",
        "Action":[
            "s3:ListBucket",
            "s3:*Object",
            "s3:GetAccelerateConfiguration",
            "s3:ListMultipartUploadParts",
            "s3:AbortMultipartUpload"
        ],
        "Resource":[
            "arn:aws:s3:::graphdb-backup-bucket",
            "arn:aws:s3:::graphdb-backup-bucket/*"
        ]
    }
]
}
```

6.11.8.2 Allow the listing of EC2 instances

The user data script that we configure later on will need permissions to list the EC2 instance where GraphDB runs. To grant the necessary permissions, paste the following JSON in the appropriate field when creating a policy:

```
{
"Version": "2012-10-17",
"Statement": [
    {
    "Effect": "Allow",
    "Action": [
        "ec2:DescribeInstances"
    ],
    "Resource": "*"
    }
]
}
```

Note: Some of those policies are needed **only** if the public TF module is used.

6.11.8.3 Allow the listing, creating, attaching, and tagging of EBS volumes

When instances start, the user data script will need to search for an available EBS volume to attach them, or if there aren't any available — create a new one. This should be performed by the user data script. Alternatively, you can create the volumes separately, attach them to the instances, and mount them to the appropriate location. To achieve this, paste the following JSON in the appropriate field when creating a policy:

```
{
"Version": "2012-10-17",
"Statement": [
    {
    "Effect": "Allow",
    "Action": [
        "ec2:CreateVolume",
        "ec2:AttachVolume",
        "ec2:DescribeVolumes"
    ],
    "Resource": "*"
    }
```

(continues on next page)

(continued from previous page)

```
]
}
```

Because the volumes need to be tagged when they are created, the EC2 instance also requires permissions for tagging the volumes. Repeat the same steps as above, but use the following JSON for the policy:

```
{
"Version": "2012-10-17",
"Statement": [
    {
    "Effect": "Allow",
    "Action": [
        "ec2:CreateTags"
    ],
    "Resource": [
        "arn:aws:ec2:*:*:volume/*",
        "arn:aws:ec2:*:*:snapshot/*"
    ],
    "Condition": {
        "StringEquals": {
        "ec2:CreateAction": [
            "CreateVolume",
            "CreateSnapshot"
        ]
        }
    }
    }
]
}
```

6.11.8.4 Allow adding of records to Route 53 private hosted zone

The user data script will also have to be able to create "A" records in a Route 53 private hosted zone. To add the needed permissions, you will first need to find the ID of your hosted zone:

1. Go to the *Route 53* dashboard
2. Select your zone
3. Expand *Hosted zone* details
4. Copy the *Hosted zone ID*

Once you have obtained your hosted zone ID, replace `<zone_id>` with the hosted zone ID in the JSON below, and paste it in the appropriate field when creating a policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "route53:GetHostedZone",
                "route53:ListResourceRecordSets",
                "route53:ListHostedZones",
                "route53:ChangeResourceRecordSets",
                "route53:ListResourceRecordSets",
                "route53:GetHostedZoneCount",
                "route53:ListHostedZonesByName"
            ],
            "Resource": "arn:aws:route53:::hostedzone/<zone_id>"
```

(continues on next page)

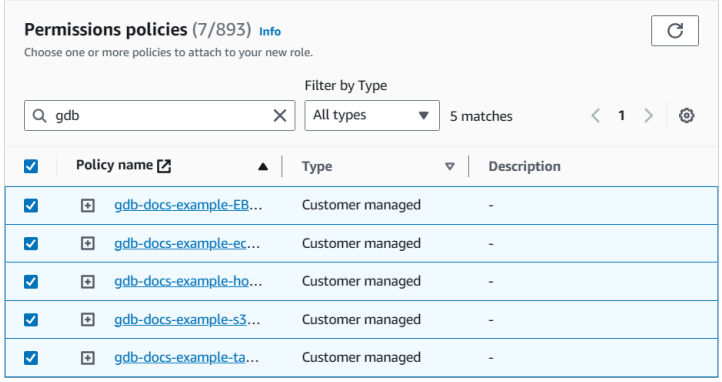
(continued from previous page)

```
        }
    ]
}
```

6.11.9 Creating an IAM role and attaching policies

Once you've created your different policies, you can create a role and associate it with the newly created policies:

1. Go to the *IAM dashboard*, and select *Roles* from the navigation menu on the left
2. Click on *Create role*
3. Select *AWS Services* ▸ *EC2* and click **"Next"**
4. Search for the policies that you created in the previous sections and select them



5. Additionally, we recommend you add a couple of optional policies:
   - The **AmazonSSMFullAccess** policy allows you to gain access to the EC2 instances using AWS Systems Manager
   - The **CloudWatchAgentServerPolicy** policy is required if you are planning on scraping the GraphDB Prometheus endpoints and pushing the metrics to Amazon CloudWatch
   - The **CloudWatchAdminPolicy** and **CloudwatchFullAccessV2** policies are required if you want to gather EC2 instance metrics which are not gathered by default, such as memory utilization metrics
6. Once you've selected all of your policies, click on *Next*
7. Add a name for the role and click on *Create role*

6.11.10 Launching GraphDB instances

6.11.10.1 Setting up the launch template

Before you can launch your GraphDB instance, you will need to create a launch template and add a security group to it.

Note:

- Launch templates describe what configurations will be used for the machines
- Security groups act like a firewall for the resources in AWS

1. Go to the *EC2 Dashboard* and select *Launch Templates* from the navigation menu on the left.
2. Click on *Create launch template* and fill in the following configurations

1. *Launch template name:* Enter a descriptive name by which to recognize your launch template
2. *Auto-scaling Guidance:* Set to **On**
3. *Application and OS images:* subscribe to the Ontotext AWS offer and select the AMI. For the purposes of this example, choose the **GraphDB Enterprise Edition (x86)** AMI as we use `r6i.2xlarge EC2` instance
4. *Instance type:* select a type such as `r6i.2xlarge`, which should be more than sufficient for a repository with one billion triples
5. *Storage (volumes) and Resource tags:* not required and should be left as is
6. Under *Network settings*, select *Create security group*
   1. Select your VPC
   2. Leave the default outbound rule set as shown unless you want to restrict it
   3. Add an inbound rule for port `7200` for the CIDR blocks that will be allowed to access GraphDB
   4. Add an inbound rule for port range `7200` (for the proxy) – `7201` (for GraphDB) and add the subnets of the load balancer
   5. Add an inbound rule for port ranges `7200-7201` and `7300-7301` and add the private subnets
   6. Add a Description
   7. Click on *Create security group*
7. Return to the Launch template creation page, refresh the list of security groups, and select the newly created security group
8. Under *Advanced details*, fill in the following configurations:
   1. *IAM instance profile:* select the role you created in the previous section
   2. At the very bottom of the form, configure the user data script that will be responsible for installing the necessary tools and GraphDB on the machines.

      Tip: An example script is available on the Ontotext-AD github repo — just be sure to replace all Terraform template variables.

      Note: If the user data script is already base64-encoded, check the box under the script field.
9. Click on the orange *Create launch template* button
10. On the *Next steps* screen that appears, click on *View launch templates*

3. Select *Auto Scaling Groups* from the bottom of the navigation menu on the left
4. Click on *Create Auto Scaling group* and fill in the following configurations
   1. *Auto Scaling group name:* Enter a descriptive name by which to recognize your auto scaling group
   2. *Launch template:* Select the launch template that was created previously and click the orange Next button
   3. *VPC:* Select the VPC that was created previously
   4. *Availability Zones and subnets:* Select all three private subnets and click Next
   5. Under *Load Balancing*, select *Attach to an existing load balancer*
   6. Select *Choose from your load balancer target groups*, then select the load balancer target group you created earlier

7. (Optional) Under *Health Checks*, you can tune the *Health check grace period*, as well as turn on the *Elastic Load Balancing health checks*, which will allow the load balancer to trigger the recreation of an instance
8. Click on *Next*
9. Under *Group Size*, enter the number of nodes in your cluster (or in this case — 3) for *Desired*, *Minimum*, and *Maximum* capacity
10. Click on *Next* until you reach the Review page
11. Click on *Create Auto Scaling group* at the bottom.

Eventually, the three new instances will be listed on the Instances screen available on the left sidebar.

6.11.10.2 Creating a cluster

In order to create the cluster, you will need to get the address of the EC2 instances.

1. Go to the *Route 53* dashboard and open you hosted zone
2. Write down all records names, for records with type "A"

Once you've written down all records names, you can create a cluster by following the *Creating and managing a cluster* documentation from one of the instances.

Tip: The recommended way to gain access to the instances is to attach the **AmazonSSMFullAccess** policy, and then use the AWS CLI to connect to an instance by its ID:

```
aws ssm start-session --target i-04d62ace38b78d994
```

After doing this, you should be able to use standard utilities like *sudo* or *su* to change the user — for example, *ubuntu*.

6.11.10.3 Opening GraphDB instances

After all instances are running, you can launch your GraphDB instance:

1. Access the *Load balancer*
2. Copy the DNS name
3. Paste it in the address bar of your browser and press **Enter**

6.11.11 Updating GraphDB configurations and versions

Because GraphDB and its configurations are baked into the AMI, you will need to recreate the EC2 instances when updating your GraphDB configuration to a newer minor version. You can do this by either manually stopping each individual instance, or scaling the cluster out, and then scaling it back in. This section describes both methods in detail.

Note: Make sure your user data script mounts the storage back to the instances.

6.11.11.1 Stopping individual EC2 instances

The faster way to update to a newer minor version and its configuration is to stop each individual EC2 instance. The downside to this method is that you are decreasing the cluster HA. In other words, if a node fails while recreating an instance, the cluster will be unable to process writes. The process is simple:

1. Update the AMI or the user data script in the launch template
2. Terminate the instances one by one, starting with the follower nodes, and leaving the leader node to be the last instance to be terminated

Note: To avoid compatibility issues, also refer to the *Migrating GraphDB configurations* documentation.

Warning: When you terminate an instance, wait for the new one to be started. Then verify that it has successfully rejoined the cluster and that it is in sync before proceeding with the next one.

6.11.11.2 Scaling the cluster out and then back in

You can also recreate the EC2 instances by scaling the cluster out and in. The advantage of this approach is that the HA will not be impacted. However, the cluster will need to replicate its state to the new nodes. This can take a significant amount of time, especially with bigger-sized repositories.

1. Update the AMI or the user data script in the launch template
2. Double the size of the cluster

   Note: Change the minimum, maximum and desired size of the auto scaling group

3. Once the new instances are started, join them to the cluster and wait until they are healthy and in sync with the cluster

   Note: Make sure to join the nodes with a single API call to avoid replicating the cluster state multiple times

4. Add scale in protection on the new nodes
5. Remove the old nodes from the cluster
6. Change the minimum, maximum and desired size of the auto scaling group to their original value
7. Remove the scale in protection

6.12 How to deploy GraphDB in Azure

GraphDB can be deployed on Microsoft Azure by following the *general installation instructions*. You can find information regarding the costs of running a GraphDB instance on the Azure website.

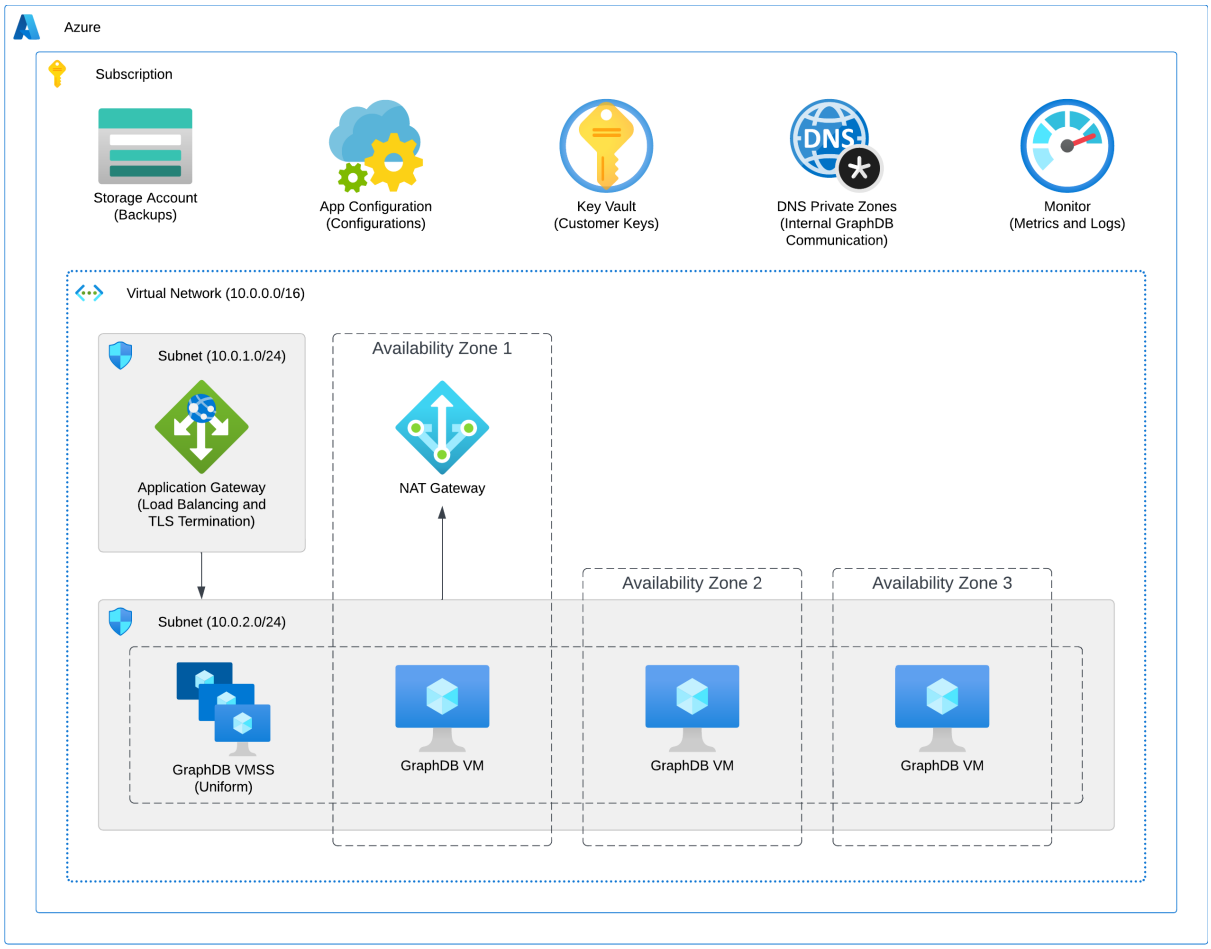
Tip: We strongly recommend using the provided Terraform module, as it automates the procedure for deploying GraphDB on Azure and mitigates challenges commonly encountered during deployment. You can find the module in our GitHub repository. The Terraform module is also available in the Terraform registry.

See also:

- *Monitoring your Azure deployment*

6.12.1 Architecture

The GraphDB architecture diagram showcases the deployment architecture for GraphDB on VM scale set instances in Azure cloud platform. The diagram illustrates the key components, and their interactions to provide a high-level understanding of the system's architecture and how it should be deployed.



Note: There are no third-party integration points on the default GraphDB deployment.

6.12.2 Prerequisites

There are several prerequisites for running a GraphDB instance on Azure:

- Access to an Azure account and an active Azure subscription.
- Active *GraphDB license* required to use the Enterprise functionalities of the database.
- Create a shell script used to initialize the VM scale set instances.
- Accept a GraphDB BYOL offer.

Note: We highly recommend you use the the GraphDB Terraform module to deploy GraphDB on Azure. The module contains Terraform scripts that fully setup the GraphDB cluster and handle DNS records registration, and

more. If you use these template files on their own, you will need to replace the placeholder values of all variables with your actual values. Please note that the scripts are template files and are therefore escaped.

6.12.2.1 Technical requirements

The following Azure services are required to complete the GraphDB deployment on Azure:

Service	Description
Resource Group	Container that holds closely related Azure resources and services that forms a solution.
Virtual Network	Private virtual network that enables Azure resources to communicate securely with each other.
VM scale set	Scalable compute service for creating and managing load balanced VM instances. Used to deploy GraphDB VM images build with GraphDB Azure Packer scripts that packages GraphDB and GraphDB external cluster proxy.
Managed disks	Block level storage for persistent data attached to VM instances. Used for persistent storage of GraphDB instance data, configurations and log files.
Application Gateway	Scalable layer 7 web load balancer managing traffic to applications in Azure. Used to load balance requests to GraphDB's external cluster proxies running in the VM scale set.
NAT Gateway	Gateway for private outbound connectivity to the internet from VM instances. Provides GraphDB VM scale set instances with NAT based internet connectivity without directly exposing them.
Key Vault	Secure storage for secret keys and certificates.
App Configuration	Service for central storage and management of application settings and feature flags. Used to store GraphDB configurations and license.
Storage Account	Secure storage for files and objects. Used for scheduled GraphDB backups that are stored as BLOBs in a storage container.
Azure Monitor	Monitoring service that collects and aggregates data, metrics and service logs from different Azure resources.
DNS Private Zones	Secure DNS service for private DNS resolution between Azure resources. Used to establish stable network identifiers for GraphDB VM scale set instances.
Public IP	Dedicated IP address exposing Azure resources on the internet. Used to expose the Application Gateway and NAT gateway on the internet.
Network Security Groups	Security rules restricting the network traffic between Azure resources in an Azure Virtual Network. Used to restrict the traffic between the Virtual Network subnets.

6.12.2.2 Required skills

Note: Deploying GraphDB on Azure requires a combination of skills in Azure infrastructure management, database administration, and system troubleshooting. Acquiring these skills may involve hands-on experience, self-study, online resources, and formal training programs provided by Azure like Azure Fundamentals or other educational platforms.

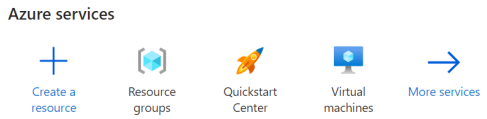
The following skills and knowledge are typically required in order to successfully deploy GraphDB on Azure:

Azure Fundamentals	Familiarity with Microsoft Azure and understanding of its core concepts, such as subscriptions and resource groups, VM instances, virtual networks, security groups and RBAC roles. Knowledge of how to navigate the Azure Portal and interact with Azure resources is essential.
Azure Virtual Networks	Understanding network fundamentals and security, subnets and network security groups. Knowledge of how to set up inbound and outbound traffic rules with NSGs to allow communication with GraphDB.
Azure VM scale sets	Proficiency in creating and managing Azure VM instances. This includes selecting the appropriate machine size, configuring security settings, managing storage (Managed Disks), and understanding VM instance lifecycle management.
Monitoring and Troubleshooting	Proficiency in monitoring the health and performance of GraphDB instances on Azure. Understanding of logging, monitoring and troubleshooting techniques using Azure Monitor, VM instance logs, and GraphDB diagnostic tools.
Linux Administration	Proficiency in Linux command-line interface (CLI) and basic administration tasks. This includes SSH access to the VM instances using Azure Bastion, navigating the file system, managing permissions, installing packages, and configuring system settings.
Database Management	Knowledge of GraphDB and its deployment requirements. Understanding of how to configure GraphDB settings, including database storage, memory allocation, and repository creation.
Database Backup and Recovery	Familiarity with backup and recovery strategies for GraphDB on Azure. Knowledge of Azure services like Storage Account for data backups and restoration processes.
High Availability and Scalability	Knowledge of implementing high availability and scalability for GraphDB on Azure. This may involve using features like VMSS Auto Scaling, load balancers, and multi-Availability Zone (AZ) deployments.
Infrastructure as Code (IaC)	Familiarity with Infrastructure as Code principles and tools like Terraform. This enables automating the provisioning and configuration of GraphDB infrastructure.
Security Best Practices	Understanding of security best practices for Azure deployments, including data encryption, access controls, identity and access management, and compliance considerations.

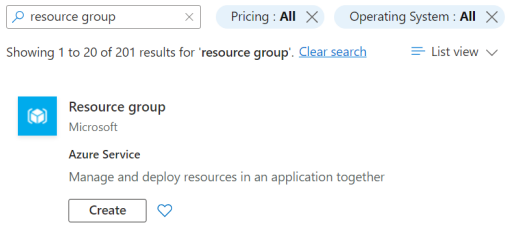
6.12.3 Creating a Resource Group

Before you start deploying your resources, you need to create a resource group to contain them.

1. Go to the *Azure Portal*.
2. Click on *Create a resource*.

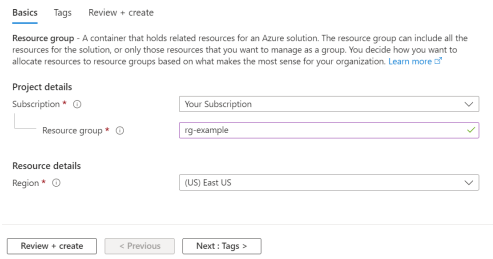


3. Search for *Resource group* and select it, then click on *Create*.



4. Fill in the required information:

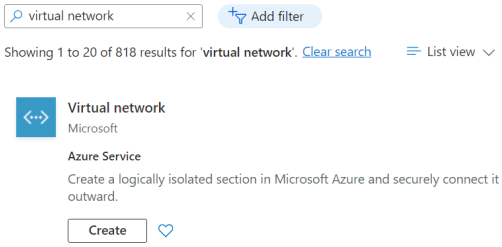
- **Resource group name:** Provide a unique name.
- **Region** (Default region is East US)



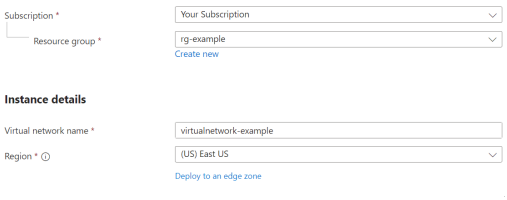
5. Click on *Review + Create*, and then on *Create*.

6.12.4 Creating a Virtual Network

1. Navigate to the Resource Group you just created.
2. Click on *+ Create*.
3. Search for *Virtual Network* and select it, then click on *Create*.



4. Fill in the required information:
   - **Name**
   - **Address Space**

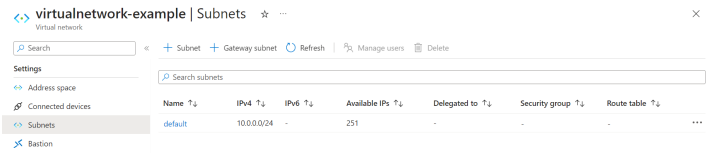


5. Click on *Review + Create*, and then on *Create*.

6.12.4.1 Creating the two subnets

Inside the Virtual Network you just created, you need to create two subnets. One of the subnets will be used for the *Virtual machine scale set*, and the other will be used for the *Application gateway*.

1. Navigate to the *Subnets* tab inside the Virtual Network you just created.

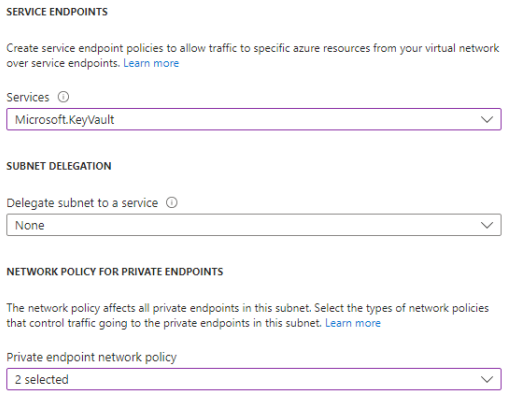


Note: You can access the Virtual Network view through the Resource Group.

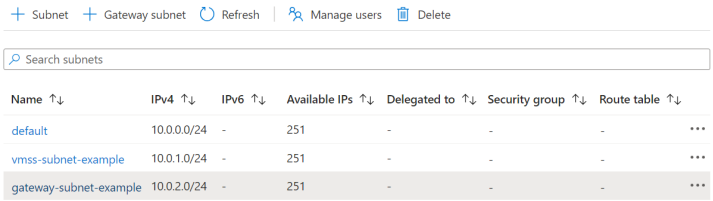
2. Click on + *Subnet*.
3. Provide a unique *Name* and *Address Range*.
4. Click on *OK*.
5. Under *Service endpoints*, search for the endpoint you need, and select it:

- When creating the **Gateway subnet**, select *Microsoft.KeyVault*.
- When creating the **VMSS subnet**, select *Microsoft.Storage*.

1. Scroll down to *Network policy for private endpoints*, and under *Private endpoint network policy*, select both *Network security groups* and *Round tables*.



2. Click *Save*, then repeat the process one more time for the second subnet.



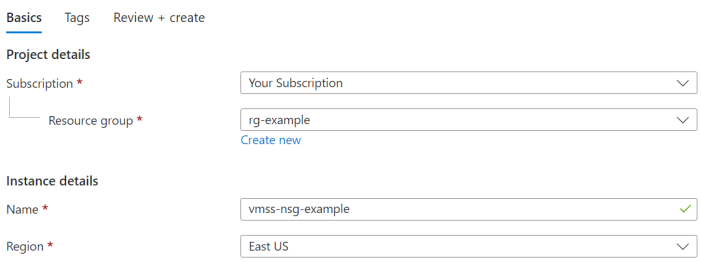
6.12.5 Creating an Application Security Groups (ASG)

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *Application Security Group* and select it, then click on *Create*.
3. Provide a unique name for your ASG.
4. Click on *Review + Create*, and then on *Create*.

6.12.6 Creating the two Network Security Groups (NSG)

You need to create two Network Security Groups (NSGs) — one for the VMSS, and one for the Gateway.

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *Network Security Group* and select it, then click on *Create*.
3. Provide a unique name for your NSG.



4. Click on *Review + Create*, and then on *Create*.
5. Repeat the process one more time for the second NSG.

After both NSGs have been deployed, you need to associate each one of them with their respective subnet.

1. Navigate to the newly deployed NSG, and go to *Subnets*.
2. Click on *Associate*.
3. Select the *virtual network you created earlier* from the drop-down menu, and then select the *respective subnet*.

6.12.6.1 Configuring the security rules for the VMSS NSG

After you create the NSGs, you also need to configure the security rules for both of them. The VMSS NSG has both Inbound and Outbound security rules. You can use the table below for reference for each individual rule.

	Source	Source configuration	Source port ranges	Destination	Destination configuration	Service	Destination port ranges	Protocol	Action	Recommended priority
Inbound rule #1	IP Addresses	**IP Address:** 10.0.0.0/16	7200-7201	Application security group	*The ASG you created.*	Custom	7200-7201	TCP	Allow	100
Inbound rule #2	Application security group	*The ASG you created.*	*	Application security group	*The ASG you created.*	Custom	7200-7201	TCP	Allow	200
Inbound rule #3	Application security group	*The ASG you created.*	*	Application security group	*The ASG you created.*	Custom	7300-7301	TCP	Allow	210
Outbound rule #1	Application security group	*The ASG you created.*	*	Service tag	**Destination service tag:** Internet	Custom	*	Any	Allow	100
Outbound rule #2	Application security group	*The ASG you created.*	*	Application security group	*The ASG you created.*	Custom	7200-7201	TCP	Allow	200
Outbound rule #3	Application security group	*The ASG you created.*	*	Application security group	*The ASG you created.*	Custom	7300-7301	TCP	Allow	210

6.12.6.2 Configuring the security rules for the Gateway NSG

After you configure the security rules for the VMSS NSG, you also need to configure the security rules for the Gateway NSG. It requires only Inbound rules. You can use the table below for reference for each individual rule.

	Source	Source service tag	Source port ranges	Destination	Destination configuration	Service	Destination port ranges	Protocol	Action	Recommended priority
Inbound rule #1	Service tag	Gateway-Manager	*	Any	N/A	Custom	65200-65535	TCP	Allow	100
Inbound rule #2	Service tag	AzureLoadBalancer	*	IP Addresses	**IP Address:** 10.0.0.0/24	Custom	8080	TCP	Allow	110
Inbound rule #3	Service tag	Internet	*	IP Addresses	**IP Address:** 10.0.0.0/24	HTTP	80	TCP	Allow	120
Inbound rule #4	Service tag	Internet	*	IP Addresses	**IP Address:** 10.0.0.0/24	HTTPS	443	TCP	Allow	130
Inbound rule #5	Any	N/A	*	IP Addresses	**IP Address:** 10.0.0.0/24	Custom	8080	Any	Deny	4000

6.12.7 Creating the two Public IP addresses

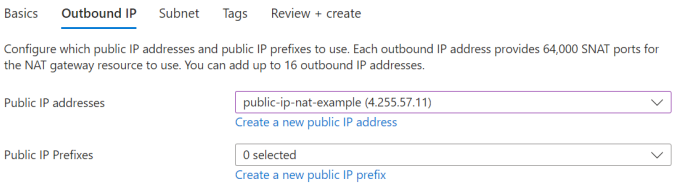
You need to create two public IP addresses — one for the Application gateway, and one for the *NAT gateway*.

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *Public IP Address* and select it, then click on *Create*.
3. Fill in the required information:
   - **Name:** Provide a unique name.
   - **Zone:** Zone redundant (recommended).
   - **Assignment:** Static or Dynamic (depending on your needs).
4. Click on *Review + Create*, and then on *Create*.
5. Repeat the process one more time for the second Public IP address.

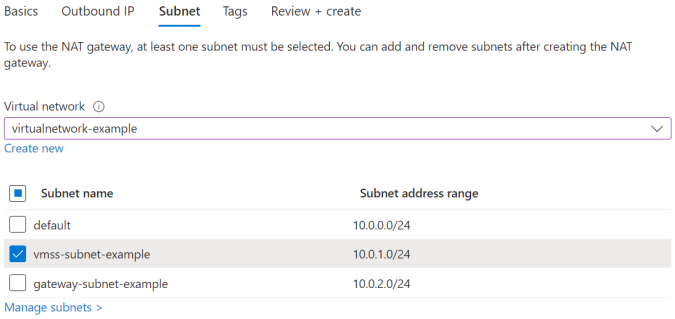
Note: The idle connection timeout is 5 minutes (the Azure default is 4 minutes). To prevent timeouts, we recommend the inclusion of Keep-Alive messages with your requests.

6.12.8 Creating the NAT gateway

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *NAT Gateway* and select it, then click on *Create* and provide a unique name.
3. Under *Outbound IP*, select the *public IP you just created*.



4. Under *Subnet*, select the *virtual network created earlier* from the drop-down menu, then select the VMSS subnet.



5. Click on *Review + Create*, and then on *Create*.

6.12.9 Creating a User-Assigned Managed Identity

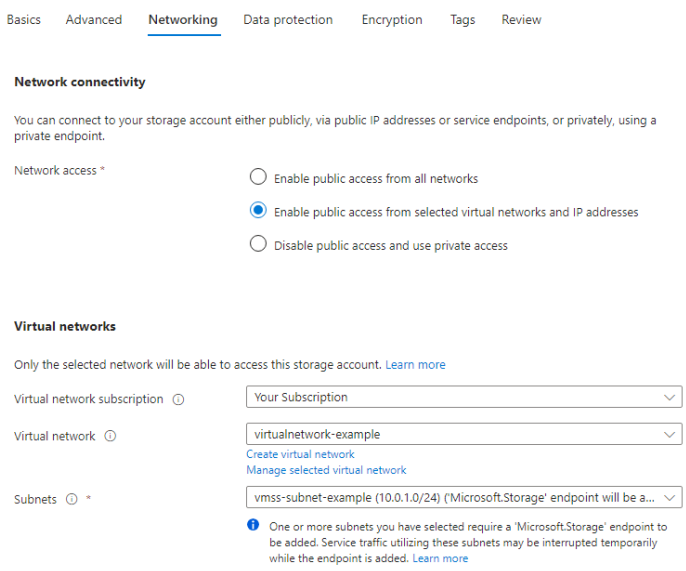
1. Navigate to your Resource Group and click on + *Create*.
2. Search for *User-Assigned Managed Identity* and select it, then click on *Create*.
3. Provide a unique name for the User-assigned managed identity.
4. Click on *Review + Create*, and then on *Create*.

Note:

- Make sure that Identity has enough rights to “Create” and “Read”
- You will need to assign identity to DNS Private Zone.

6.12.10 Creating the Storage Account

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *Storage Account* and select it, then click on *Create*.
3. Under *Networking*, select the virtual network and choose the *VMSS subnet*.



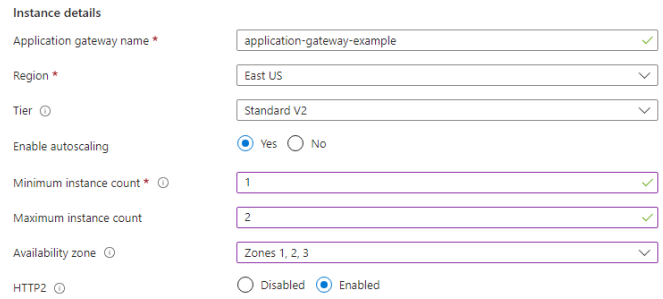
4. Under *Data Protection*:
   - Check *Enable soft delete for blobs*, with *Days to retain deleted blobs:* 31.
   - Check *Enable soft delete for containers*, with *Days to retain deleted containers:* 31.
   - Check *Enable blob change feed*, and select *Delete change feed logs after (in days):* 31.
   - Uncheck everything else.
5. Click on *Review + Create*, and then on *Create*.

After the Storage Account has been deployed:

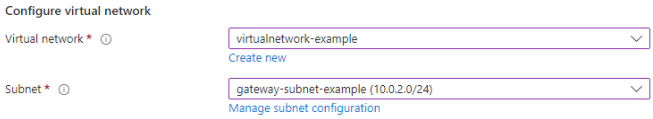
1. Navigate to the newly created Storage Account, and go to *Containers*.
2. Create containers for backup.

6.12.11 Creating the Application Gateway

1. Navigate to your Resource Group and click on *+ Create*.
2. Search for *Application Gateway* and select it, then click on *Create*.
3. Set the following instance details:
   - **Minimum instances:** 1
   - **Maximum instances:** 2
   - **Availability zone:** Select all 3 availability zones.



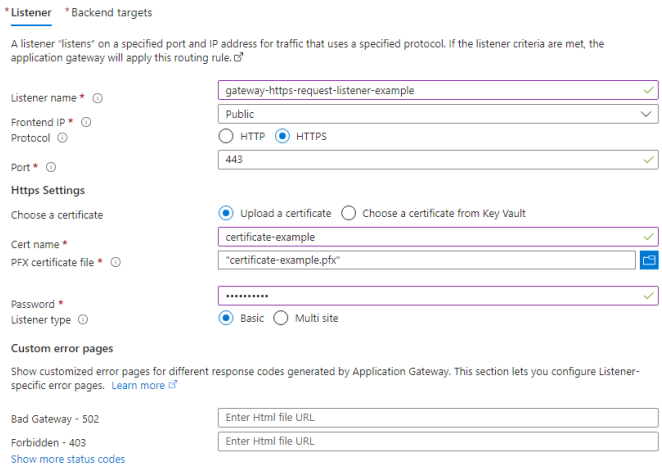
4. Under *Configure virtual network*, select the gateway subnet.



5. Under *Frontends*, select the *gateway IP created earlier*.
6. Under *Backends*, add a backend pool with targets.

> **Note:** Once you create and properly configure the *VMSS*, Azure should automatically add it as a target to the backend pool.

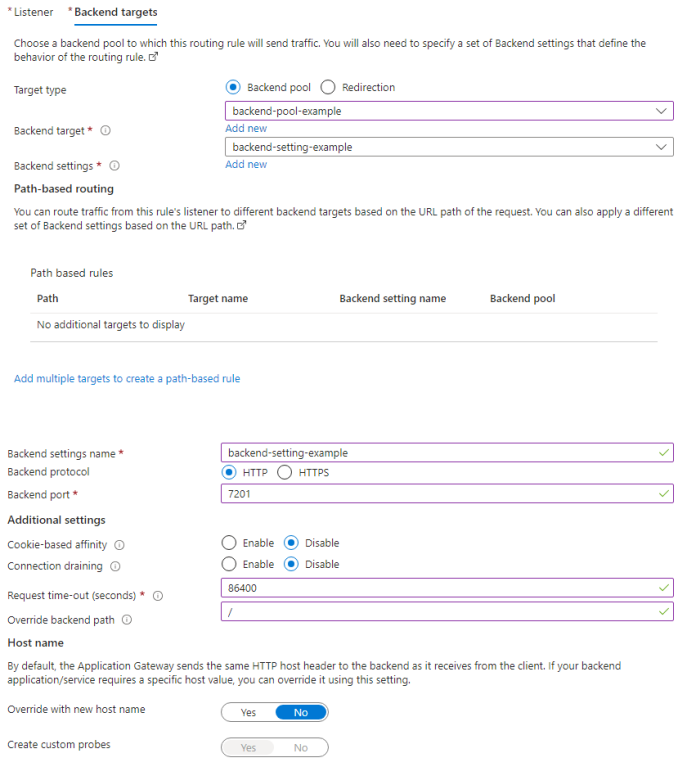
7. Under *Configuration*, define the *Rule*, *Listener* and *Backend targets*.
   - Configurations for the **Listener:**
     - *Protocol:* HTTPS
     - *Port:* 443
     - *Certificate:* Upload your certificate.



   - Configurations for the *Backend targets*:
     - Select the **Backend pool** you just created, and add new settings, as follows:
     - *Protocol:* HTTP
     - *Port:* 7201
     - *Request time-out:* 86400.

       > **Note:** 86400 is the maximum number of seconds allowed. Such a long time is necessary for GraphDB to be able to operate.

     - *Override backend path:* /
     - *Override with new host name:* No



8. Set a high priority for the rule.

> **Tip:** We recommend a value of 100.

9. Review and create the resource.

> **Warning:** Deploying the application gateway can take longer than most of the other resources.

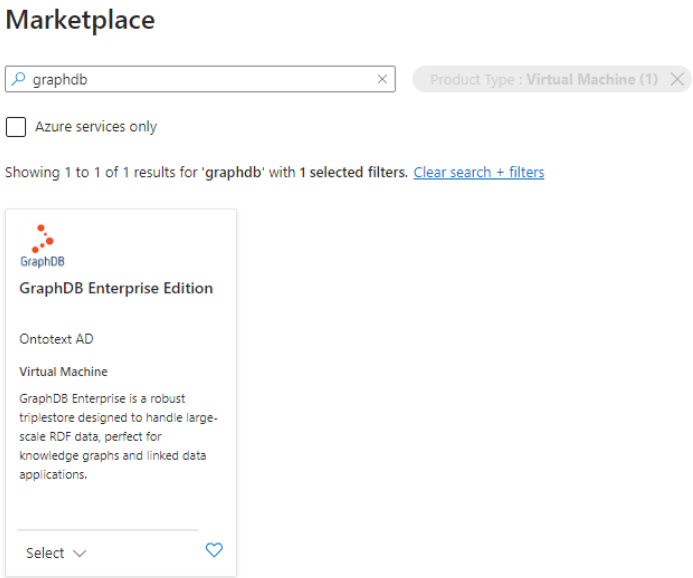
6.12.12 Creating the Virtual Machine Scale Set (VMSS)

Now that all other required resources have been deployed, the next step is to create the Virtual Machine Scale Sets (VMSS), where each instance functions as a cluster node. VMSS enables the deployment and management autoscaled, identical virtual machines, allowing you to dynamically scale in or out.

Additionally, this is where you can add a custom user data script if you want to use one.

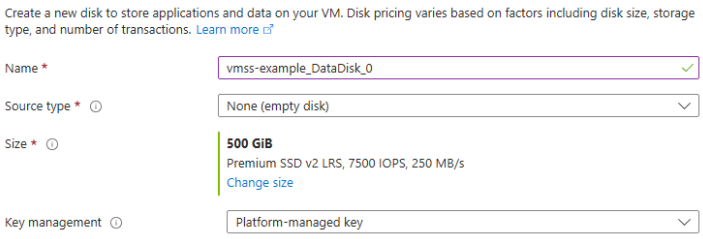
> **Tip:** You can use the custom user data script that is part of the Terraform module for reference when creating your script.

1. Navigate to your Resource Group and click on + *Create*.
2. Search for *Virtual Machine Scale Set* and select it, then click on *Create*.
3. Under *Scale set details*, provide a unique name and select all necessary availability zones.
4. Under *Orchestration*, choose Uniform.
5. Under *Instance details*, click on *See all images* and search for the GraphDB image in the *Marketplace*.



6. For each availability zone, add a managed disk — the recommended disk is Premium SSDv2 with 7500 IOPS and 250 throughput, 500GB.

> **Warning:** At the end of the deployment process you must attach each a disk to each VMSS instance as LUN 2.



7. Add to the earlier created *virtual network* and select the VMSS subnet.
   - Under *Load balancing options*, select the *Application Gateway* created earlier.
8. Under *Management*, add the *user-assigned managed identity* you created earlier.
9. If you want to use a cluster, we recommend a value of 3, 5 or 7 under *Scaling*.
10. Enable health monitoring with the following configuration:
    - *Type:* Application health extension
    - *Protocol:* HTTP
    - *Port number*: 7200
    - *Path:* /rest/cluster/node/status
11. If you want to use a user data script, you can add it under *Advanced*.

> **Tip:** You can use the custom user data script that is part of the Terraform module available on the Ontotext-AD github repo for reference when creating your script.

12. Review and create the resource.

6.12.12.1 Attaching the managed disks

After you have deployed your VMSS, you need to attach a disk to each of your VMSS instances.

Hint: The disk management script from the Terraform module can be used as a reference for attaching your managed disks.

1. Navigate to the newly created VMSS, and go to *Instances*.
2. Open the instance you want to attach a managed disk to.
3. Under *Disks*, click on *Attach existing disks*.
4. Add the disk you created earlier with *LUN* 2.

6.12.13 Setting up and managing your cluster

Once all the resources are installed and configured correctly, you can create a cluster by accessing Bastion and using the REST API.

Warning: If one of your instances dies or becomes unavailable, you will need to manually remove the node and add a new replacement node.

See also:

- *Overview of clusters*
- *Creating and managing a cluster*
  - *Using the REST API*
  - *Manage cluster membership*
- Azure Bastion documentation

6.12.14 Accessing the Workbench through the Application Gateway

After all the resources are installed, the cluster has been created and all the instances are healthy, the Workbench can be accessed from the Application Gateway.

Warning: You cannot use the Workbench interface to create clusters in an Azure deployment. If you haven't created a cluster yet, Workbench would not be loaded.

6.12.15 Scaling out and back in VMSS

1. Navigate to the *Resource group* containing your VMSS in the *Azure Portal*.
2. Go inside the *VMSS* and select *Scaling* in the navigation pane.
3. Under *Configure*, choose *Manual scale* or *Custom autoscale* based on your preference.
4. Under *Scale-in Policy*, select the *NewestVM - Balance across the availability zones, and delete the newest created virtual machines*.
5. Adjust the *Instance count* to the desired number of instances.

Note: If you require a GraphDB Cluster, the instance count is recommended to be 3, 5 or 7.

6. Click on *Save* to apply the scaling configuration.

6.12.16 Required manual maintenance of instances

If the Terraform script is not used, some manual maintenance of the instances will be required.

In case of a cluster failure, once the VMs are recreated, you will need to manually handle:

- Volume attaching and mounting
- DNS address updates
- Provisioning of GraphDB configurations and backup scripts.

6.13 How to deploy GraphDB in GCP

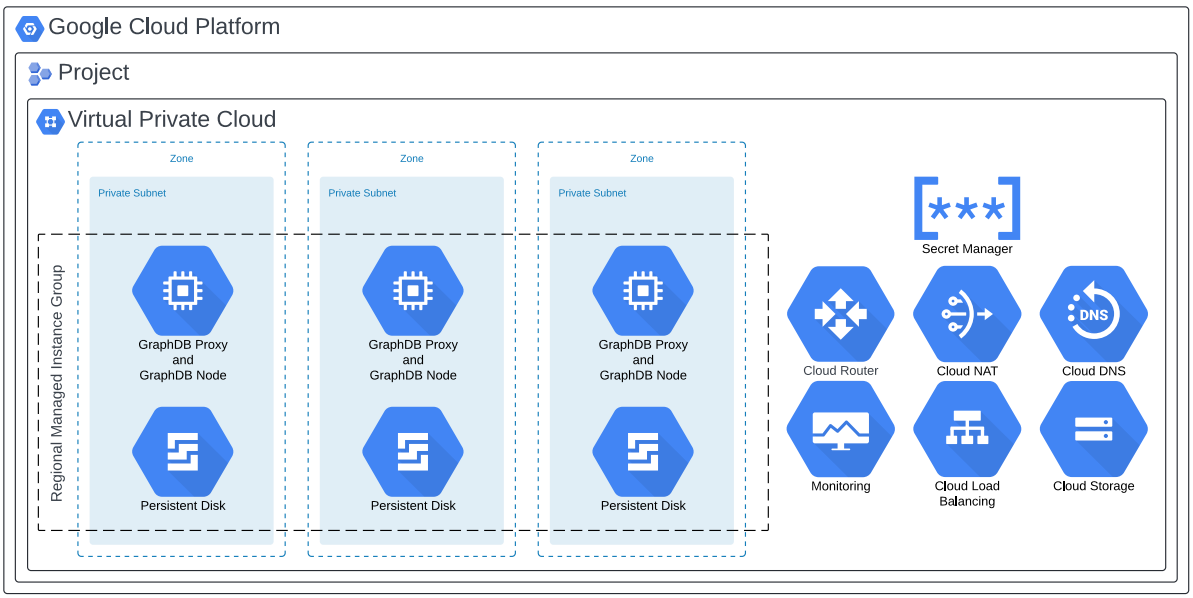
GraphDB can be deployed on Google Cloud Platform (GCP) by following the *general installation instructions*. You can find information regarding the costs of running a GraphDB instance on the Google Cloud Platform website.

This documentation will walk you through the process of setting up the necessary environment for deploying GraphDB on Google Cloud Platform.

Note: Ontotext maintains a Terraform module that automates the entire procedure of deploying GraphDB on Google Cloud Platform. Learn more about how to use it at our GitHub repository.

6.13.1 Architecture

The GraphDB architecture diagram showcases the deployment architecture for GraphDB on Google Compute Engine instance in Google Cloud Platform. The diagram illustrates the key components, and their interactions to provide a high-level understanding of the system's architecture and how it should be deployed. There are no third-party integration points on the default GraphDB deployment.



6.13.2 Prerequisites

There are several prerequisites for running a GraphDB instance on GCP:

- Active *GraphDB license* required to use the Enterprise functionalities of the database
- Access to an GCP account

Note: We recommend the use of an Identity and Access Management user for the deployment instead of a root user account.

6.13.2.1 Technical requirements

The following Google Cloud Platform services are required to complete the GraphDB deployment on GCP:

Service	Description
Google Compute Engine (GCE)	Server instance that are used to host the database application
Cloud Balancing	Used for load balancing the GraphDB cluster nodes
Google Cloud Persistent Disk	Persistent Disk volumes are used for storing the data
Google Cloud Identity and Access Management (IAM)	Provides user and access management for your GraphDB deployment
Google Cloud Secrets Manager	Used to store various GraphDB configurations
Google Cloud Storage	Used to store GraphDB Backups
Cloud Monitoring	Used to monitor the status of the GraphDB cluster

6.13.2.2 Required skills

The following skills and knowledge are typically required in order to successfully deploy GraphDB on GCE Instance:

Google Cloud Platform Fundamentals	Familiarity with Google Cloud Platform (GCP) and understanding of its core concepts, such as Google Cloud Engine instances, security, VPCs and IAM roles. Knowledge of how to navigate the Google Cloud Platform Web Console and interact with GCP services is essential.
Google Cloud Engine Instance Management	Proficiency in creating and managing Google Cloud Engine instances. This includes selecting the appropriate instance type, configuring security settings, managing storage (Cloud Persistent Disk Volumes), and understanding Google Cloud Engine instance lifecycle management.
Networking and Security	Understanding of networking concepts in GCP, including VPC (Virtual Private Cloud) configuration, subnets, routing tables, and firewall rules. Knowledge of how to set up inbound and outbound traffic rules to allow communication with GraphDB.
Linux Administration	Proficiency in Linux command-line interface (CLI) and basic administration tasks. This includes SSH access to GCE instances, navigating the file system, managing permissions, installing packages, and configuring system settings.
Database Management	Knowledge of GraphDB and its deployment requirements. Understanding of how to configure GraphDB settings, including database storage, memory allocation, and repository creation.
Database Backup and Recovery	Familiarity with backup and recovery strategies for GraphDB on GCP. Knowledge of GCP services like Google Cloud Storage for data backups and restoration processes.
Monitoring and Troubleshooting	Proficiency in monitoring the health and performance of GraphDB instances on GCP. Understanding of logging, monitoring and troubleshooting techniques using GCP Cloud Monitoring, GCE instance logs, and GraphDB diagnostic tools.
High Availability and Scalability	Knowledge of implementing high availability and scalability for GraphDB on GCP. This may involve using features like GCP Auto Scaling, load balancers, and multi-zone deployments.
Infrastructure as Code (IaC)	Familiarity with Infrastructure as Code principles and tools like Terraform. This enables automating the provisioning and configuration of GraphDB infrastructure on GCP.
Security Best Practices	Understanding of security best practices for GCP deployments, including data encryption, access controls, identity and access management, and compliance considerations.

6.13.3 Setting up your Virtual Private Cloud (VPC)

1. Go to the *VPC Networks* and click on *Create VPC Network*.
2. Under *Subnet Creation mode*, select the *Automatic* option. This will allow you to configure the subnets in the regions.
3. Provide a descriptive *Name* by which to recognize your VPC.
4. Use the table below as a reference to define the *Firewall Rules* to allow inbound connections to the GraphDB instance.

Name[1]	Type	Targets	Filters	Protocols / ports	Action	Priority[2]
rule-allow-custom	Ingress	Aply to all	IP ranges: 10.128.0.0/9	all	Allow	65,000
rule-allow-icmp	Ingress	Aply to all	IP ranges: 0.0.0.0/0	icmp	Allow	65,000
rule-allow-rdp	Ingress	Aply to all	IP ranges: 0.0.0.0/0	tcp:3389	Allow	65,000
rule-allow-ssh	Ingress	Aply to all	IP ranges: 0.0.0.0/0	tcp:22	Allow	65,000
rule-deny-all-ingress	Ingress	Aply to all	IP ranges: 0.0.0.0/0	all	Deny	66,000
rule-allow-all-egress	Egress	Aply to all	IP ranges: 0.0.0.0/0	all	Allow	66,000

5. Click on *Create VPC*

Tip: It may take a while for the VPC to deploy. You will know that it is done deploying when you the message "Successfully created network" appears on the bottom of your screen.

6.13.4 Setting up your Cloud DNS private hosted zone

The GraphDB Raft implementation requires static addresses. This is achieved by creating a private hosted zone in Google Cloud DNS and registering the instances there.

1. From the *Cloud DNS dashboard*, click on *Create DNS Zone*.
2. Under *Zone type*, select *Private*.
3. Provide a *Zone name*.
4. Provide a *DNS name* such as `graphdb.cluster`.
5. Under the *Networks* section select the networks from which you want to have access to the hosted zone.
6. Click *Create*

Note: *Later,* you will also need to create "A" records for the instances.

6.13.5 Creating a Cloud Storage Bucket

Tip: This step is **optional**, but recommended.

GraphDB can store backups to Cloud Storage and, if needed, restore from them.

To create an Cloud Storage bucket:

1. From the *Cloud Storage console*, click on *Create*.
2. Provide a globally unique name.
3. Select your region, scroll down and click *Create bucket*.

[1] The names in the table are illustrative and describe what the rule does. Use names that make sense for your deployment.

[2] The priorities given in the table are illustrative. What is important is that the priority of rules `rule-deny-all-ingres` and `rule-allow-all-egress` is higher than of the rest of the rules.

In order to block all public access when you create a bucket you should tick Enforce public access prevention on this bucket on the *Choose how to control access to objects section*.

6.13.6 Importing a certificate into GCP Certificate Manager

Tip: This step is **optional**.

While serving GraphDB requests over a secured and encrypted connection is not strictly required, it is nevertheless highly recommended. This section goes over the process of importing a certificate into GCP Certificate Manager so that you can use that certificate later when *creating the load balancer*.

1. Go to the *Certificate Manager* console and click on *Create*.
2. Paste or upload your *Certificate*.
3. Paste or upload your *Private Key*.

   Warning: You need to remove the password for the key before pasting it.

4. You can optionally add labels, then click on *Create*

6.13.7 Setting up the Load Balancer

1. Go to *Load Balancing service* and click on *Create load balancer*.
2. Click on *Network Load Balancer* and provide the following configurations:
   - *Type of load balancer*: Network Load Balancer
   - *Proxy or passthrough*: Proxy load balancer
   - *Public facing or internal*: Public facing (external)

     Tip: Otherwise GraphDB will not be accessible externally.

   - *Global or single region deployment*: Best for global workloads
   - *Load balancer generation option*: Global external proxy Network Load Balancer
3. Then click on the *Configure* button.

On the next screen, provide a unique *Load Balancer name*.

1. Select your *Region* from the *Region* dropdown field.
2. Choose the *Network* in which you want to deploy the load balancer.

Under *Backend configuration*:

1. *Backend type*: Select *Instance group*.
2. *Protocol*: Select *TCP*.
3. Under *Instance Groups*, configure the following parameters:

- *Instance group*: Select the instance group that you used to deploy the GraphDB nodes.
- *Port numbers*: 7200

1. Under *Healthcheck configuration*, configure the following parameters:
   - *Protocol:* HTTP

- *Port:* 7201
- *Request path:* /rest/cluster/node/status

2. Under *Frontend configuration*, configure the following parameters:
   - *Name*: Provide a unique name.
   - *Protocol*: SSL
   - *Port*: 443
   - *Certificate*: Choose the certificate that you have created in the previous section.
3. Click *Create*.

6.13.8 Setting up service account

The GCE instances require certain permissions in order to perform several tasks in GCP. This section describes the permissions they need, what they are used for, and how to create them.

To achieve all of this, create an instance profile and then attach it to the instances:

1. Go to the *Identity and Access Management* (IAM) dashboard and select *Service accounts* from the navigation menu on the left.
2. On the service account details provide the following configurations:
   - Define name for the *Service account name*.
   - Define *Service account description*.
3. On the *Grant this service account access to project (optional)* step you can specify the role for the service that you want to access from the GCE instances.
4. On the *Grant users access to this service account (optional)* step you can specify Service account user roles or Service account admin roles that can access the service account.
5. Click on *Done*.

Note: If you are planning to store GraphDB backups to Cloud Storage, the GCE instance needs to be able to read, write, and list objects in Cloud Storage service. Perform the steps from *Setting up service account*, but under *Grant this service account access to project*, select *Cloud Storage Admin Role*.

6.13.9 Launching GraphDB instances

Before you can launch your GraphDB instance, you will need to create an instance template with an existing GraphDB image. You can do it by using the following command using the gcloud command line interface:

```
gcloud beta compute instance-templates create graphdb-instance-template \
--project=example-project-name --machine-type=n2-highmem-8 \
--network-interface=network=default,network-tier=PREMIUM,stack-type=IPV4_ONLY \
--instance-template-region=us-central1 --maintenance-policy=MIGRATE \
--provisioning-model=STANDARD  \
--scopes=https://www.googleapis.com/auth/cloud-platform \
--create-disk=auto-delete=yes,boot=yes,device-name=graphdb-instance-template,image=projects/graphdb-
↪public/global/images/ontotext-graphdb-10-7-6-202410151959,mode=rw,size=20,type=pd-balanced \
--create-disk=device-name=persistent-disk-1,mode=rw,size=500,type=pd-balanced \
--no-shielded-secure-boot --shielded-vtpm --shielded-integrity-monitoring \
--reservation-affinity=any
```

6.13.9.1 Setting up Instance Group

In order to create an instance group based on the instance template that you have created in the previous section, execute the following steps in the GCP Web Console.

1. Go to Compute Engine Service.
2. On the left tab select *Instance Groups* option.
3. Click on *Create* button and fill the fields below.
   - *Name*: Define a name that you want to use for the instance template.
   - *Description (Optional)*: Provide a description to help easily identify the deployed resources.
   - *Instance template*: Select the instance template that you have created via the gcloud CLI, as described in *Launching GraphDB instances*.
   - *Location*: Choose *Multiple zones* and check if the three zones in the region you selected are selected (they should be selected by default).
   - *Autoscaling*: Set both *Minimum number of instances* and *Maximum number of instances* to 3.
   - *Port Mapping*: 7200
4. Click on the *Create* button

6.13.9.2 Creating a cluster

In order to create the cluster, you will need to get the address of the GCE instances.

1. Go to the *Cloud DNS* dashboard and open your private hosted zone.
2. Write down the names of all records with type "A".

Once you've written down all records names, you can create a cluster by following the *Creating and managing a cluster* documentation from one of the instances.

Tip: The recommended way to gain access to the instances is by going to *Compute Engine* and selecting the instance you want to access, then clicking on *SSH*.

6.13.9.3 Opening GraphDB instances

Once all instances are running, you can launch your GraphDB instance:

1. Access the *Load balancer*.
2. Copy the DNS name.
3. Paste it in the address bar of your browser and press `Enter`.

6.13.10 Updating GraphDB configurations and versions

Since GraphDB and its configurations are baked into the Instance Image, you will need to recreate the GCE instances when updating your GraphDB configuration to a newer minor version. You can do this by either manually stopping each individual instance, or scaling the cluster out, and then scaling it back in. This section describes both methods in detail.

6.13.10.1 Stopping individual GCE instances

The faster way to update to a newer minor version and its configuration is to stop each individual GCE instance. The downside to this method is that you are decreasing the cluster HA. In other words, if a node fails while recreating an instance, the cluster will be unable to process writes. The process is simple:

1. Update the Instance Image in the instance template.
2. Terminate the instances one by one, starting with the follower nodes, and leaving the leader node to be the last instance to be terminated.

See also:

To avoid compatibility issues, also refer to the *Migrating GraphDB configurations* documentation.

> **Warning:** When you terminate an instance, wait for the new one to be started. Then verify that it has successfully rejoined the cluster and that it is in sync before proceeding with the next one.

6.13.10.2 Scaling the cluster out and then back in

You can also recreate the GCE instances by scaling the cluster out and in. The advantage of this approach is that the HA will not be impacted. However, the cluster will need to replicate its state to the new nodes. This can take a significant amount of time, especially with bigger-sized repositories.

1. Update the Instance Image in the launch instance template.
2. Double the size of the cluster.

   > **Note:** Change the minimum, maximum and desired size of the auto scaling group.

3. Once the new instances are started, join them to the cluster and wait until they are healthy and in sync with the cluster.

   > **Note:** Make sure to join the nodes with a single API call to avoid replicating the cluster state multiple times.

4. Add scale in protection on the new nodes.
5. Remove the old nodes from the cluster.
6. Change the minimum, maximum and desired size of the auto scaling group to their original values.
7. Remove the scale in protection.

6.14 Using GraphDB's LLM tools with external clients

The Large Language Model (LLM) tools that enable features such as *Talk to Your Graph* can be used outside of GraphDB with external clients. The tools are exposed through the API and can be used with either Model Context Protocol (MCP)-compatible clients or with frameworks such as Dify.

6.14.1 Using GraphDB's LLM tools with MCP clients

The Model Context Protocol (MCP) is an open protocol that standardizes how applications provide context to large language models (LLMs). It provides a standardized way to connect AI models to different data sources and tools, including the LLM tools of GraphDB, and build agents and complex workflows. The MCP protocol is designed to support streaming communication where the client sends messages and the server responds with events in a structured format, one per message. Each message is sent as a `data:` block in a separate event, just like a chunked stdio or websocket stream. The MCP responses are wrapped as valid SSE events, preserving the MCP structure.

GraphDB runs an MCP server that enables MCP-compatible clients to perform semantic and structured queries against RDF repositories using GraphDB. It is designed to serve requests over the streamable HTTP transport, providing streaming compatibility across various clients. Communication through the transport is done through HTTP POST and GET requests.

The GraphDB MCP server runs on the same HTTP port as GraphDB. The GraphDB MCP server exposes an `/mcp` endpoint for `POST` and `GET` requests.

See also:

The MCP server can be configured through several configuration properties, which control the allowed number of concurrent sessions and their duration. These properties are described in described in details in the *MCP server properties* documentation.

6.14.1.1 MCP security configurations

The GraphDB MCP server uses Spring Security to define basic access permissions. By default, both endpoints are open to anonymous users with access `READ`. This means any client can connect to these endpoints, as long as the free access is enabled with at least `READ` given repository rights. This is acceptable only if the underlying GraphDB instance allows read access to the target repository, either because the user is authenticated or GraphDB is configured for free access.

GraphDB's built-in *GraphDB access control* rules are still enforced at the repository level. This ensures that only users or API clients with the appropriate permissions (such as the ability to read a given resource) can query specific data or graphs in the repositories, and that unauthorized access is blocked by GraphDB itself, and not just at the HTTP level.

6.14.1.2 Exposed LLM tools for MCP clients

GraphDB exposes all query methods available when creating *Talk to Your Graph* agents as tools, as well as a few additional others:

- **Repository listing**: Provides system information about available repositories in GraphDB MCP Server.
- **Full-text search**: Queries GraphDB using *Full-text search* and returns a subgraph of RDF triples.
- **IRI discovery**: *Discovers IRIs* based on full-text search label matches.
- **Retrieval search**: Queries GraphDB by full-text search and returns textual data suitable for summarization or memory operations. Optimized for ChatGPT Retrieval.
- **Autocomplete IRI discovery**: Uses GraphDB's autocomplete index to fetch relevant IRIs by searching their names.
- **Similarity search**: Finds conceptually related entities using a text similarity index.
- **SPARQL query interface**: Executes a SPARQL `SELECT`, `CONSTRUCT`, or `DESCRIBE` query against GraphDB and return the results.
- **Ontology schema extraction**: Extracts ontology schemas used by clients to build structured queries.
- **Similarity options discovery**: Provides information about available similarity search capabilities in a GraphDB repository, including supported vector fields, connector types, and similarity indexes. Clients use this information to correctly configure parameters when performing similarity searches.

6.14.1.3 SSE transport and the MCP gateway

The GraphDB MCP server also supports communication over the Server-Sent Events (SSE) for backward compatibility with older clients. The use of SSE can be configured through the `graphdb.mcp.transport.mode` configuration property, which is described in details in the *MCP server properties* documentation.

To handle the full bidirectional behavior required by the MCP protocol, the GraphDB MCP server exposes two endpoints for SSE communication:

- POST `/mcp/message` for the client to server communication
- GET `/mcp/sse` for the streaming of responses from the server to the client

Note: Prior to GraphDB 11.3, the MCP server supported only the SSE protocol, and did so by default.

Communication through the SSE is facilitated by an MCP gateway, which allows MCP clients to connect to a secured GraphDB MCP server. Some older MCP clients support only `stdio`-based MCP servers, and thus a midleware `mcp-gateway` service is required to bridge the gap between `stdio` employed by those MCP clients, and `http/sse` used by the GraphDB MCP server. The MCP gateway converts `stdio` into `http/sse` and routes it to the GraphDB MCP server.

Tip: GraphDB maintains a fork of the MCP gateway service, with its own installation and configuration instructions.

6.14.2 Using GraphDB's LLM tools with Dify and other LLM agent frameworks

GraphDB also exposes LLM tools using the OpenAPI specification for integration with LLM agent frameworks such as Dify. , custom UIs, and Python-based agents. This API facilitates the use of *Talk to Your Graph* by bridging the gap between rapid prototyping in the Workbench and production-grade orchestration with external agents.

6.14.2.1 Supported configuration types

Unlike with MCP clients, tools exposed to these frameworks are exposed on a per-repository basis, using the base path `/rest/llm`. The exposed tools can be configured in one of two ways:

- Static — Static tool definitions configured in `yaml`
- Dynamic — Tools defined in the preconfigured agents in *Talk to Your Graph*

Tip: If you have already *configured a Talk to Your Graph agent*, you can obtain the configuration ID of that agent by navigating to *Talk to Your Graph* ▸ *Edit agent settings* of the agent you want to use and then selecting *External integration configuration* on the *Edit Agent* dialog box.

6.14.2.2 Exposed tools for the Dify framework

GraphDB exposes all query methods available when creating *Talk to Your Graph* agents as tools. Since tools are configured on a per repository level, the **Repository listing** and **Ontology schema extraction** *tools exposed to MCP clients* are not needed and thus not available.

The full list of tools can be obtained by the client with the following call, where `{configType}` is either `yaml` or `ttyg`, and `configId` is the unique identifier of the LLM tool configuration:

```
GET /rest/llm/tool/{configType}/{configId}
```

Tip: See the *Example yaml configuration* below.

You can access tools with the following call, where `{toolType}` is the name of the tool:

```
POST /rest/llm/tool/{configType}/{configId}/{toolType}
```

The name of the tools are:

- `sparql_query`
- `fts_search`
- `similarity_search`
- `retrieval_search`
- `autocomplete_iri_discovery_search`
- `iri_discovery`
- `now`

6.14.2.3 Example yaml configuration

The following `yaml` configuration enables the SPARQL query tool and full-text search for a repository named `acme`, and the SPARQL query tool for a repository named `starwars`:

```yaml
acmeConfig:
  repositoryId: "acme"
  tools:
    sparql_query:
      enabled: true
      ontologyGraph: "http://example.com/acme"
      ontologyQuery: null
      addMissingNamespaces: false
    retrieval_search:
      enabled: false
      connectorInstance: null
      queryTemplate: "{\n  \"query\": {\n    \"type\": \"string\",\n    \"title\"\
        : \"Query\"\n  },\n  \"filter\": {\n    \"type\": \"object\",\n    \"properties\"\
        : {\n      \"document_id\": {\n        \"type\": \"string\",\n        \"title\"\
        : \"Document Id\"\n      }\n    }\n  },\n  \"top_k\": {\n    \"type\": \"\
        integer\",\n    \"title\": \"Top K\",\n    \"default\": 3\n  }\n}"
      limit: 0
    similarity_search:
      enabled: false
      similarityIndex: null
      resultThreshold: 0.6
      limit: 0
    fts_search:
      enabled: true
      limit: 0
    iri_discovery:
      enabled: false
starwarsConfig:
  repositoryId: "starwars"
  tools:
    sparql_query:
      enabled: true
      ontologyGraph: "https://swapi.co/ontology/"
      ontologyQuery: null
      addMissingNamespaces: false
```

6.14.2.4 Using Dify API extension

You can also enable compatibility with Dify's tool-calling protocol, which supports seamless plug-in of GraphDB tools into Dify workflows:

```
POST /rest/llm/{configType}/{configId}/dify
```

When creating an agent in Dify, this extension allows you to obtain the SPARQL instructions or ontology directly from a static `yaml` config file or from the configurations of a *Talk to Your Graph agent*.

- To obtain static instructions, simply point to the `yaml` file you want to use, for example:

  ```
  POST /rest/llm/yaml/acmeConfig/dify
  ```

- To obtain dynamic instructions from a Talk to Your Graph agent, use the Dify extension endpoint you can find by navigating to *Talk to Your Graph* ‣ *Edit configurations* of the agent whose instructions or ontology you want to use, and then navigating to *External integration configuration* ‣ *Dify extension*.

You can then use that endpoint by creating a new variable in the agent (which can take either `sparql_instructions` or `ontology`), and adding the endpoint to that variable.

CHAPTER

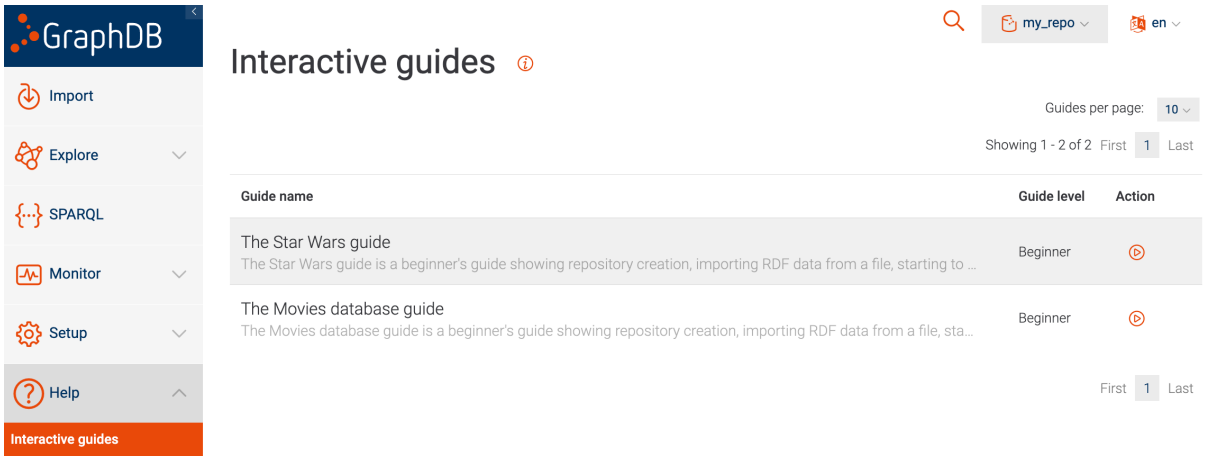
SEVEN

TUTORIALS

7.1 Running the interactive user guides

GraphDB includes a set of interactive tutorials that will walk you through key GraphDB capabilities using the Workbench user interface. They can be accessed from *Help* ▸ *Interactive guides*, as well as via the *Take me to the guides* button in the center panel of the GraphDB Workbench startup screen.

Tip: Learn more about the W3C Knowledge Graph standards and software development with GraphDB for free at the Graphwise Academy.



7.1.1 Available guides

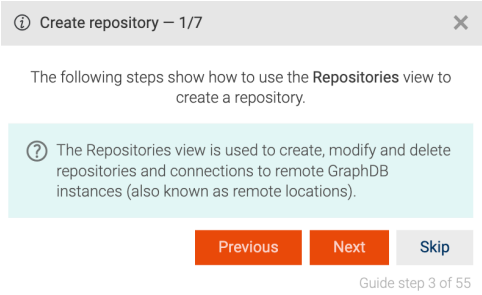
Each guide has a name, a description, a level (Beginner, Intermediate, or Advanced), and a *Run the guide* button, which starts the tutorial. Currently, GraphDB 11.3 offers two such tutorials:

- The **Star Wars** guide: Designed for beginners and using the Star Wars dataset, which you can download within the guide, this tutorial will walk you through some basic GraphDB capabilities such as *creating a repository*, *importing RDF data from a file* in it, and exploring the data through the *Visual graph*.
- The **Movies database** guide: Also designed for beginners and using a dataset with movie information, this tutorial will show you some additional capabilities like exploring your data from the *class hierarchy* perspective, some SPARQL queries, as well as exploring RDF through the tabular view.

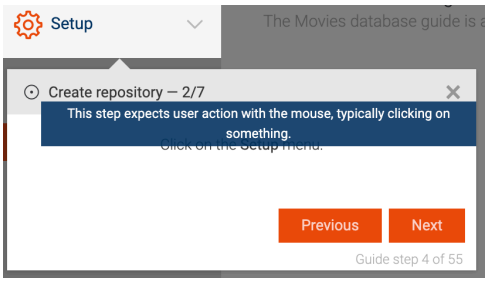
7.1.2 Running a guide

To start a guide, click *Run* ⊳. This will activate a series of dialog boxes that will guide you through the steps of the tutorial. While the guide is running, the remaining part of the Workbench remains darker and is inactive.

Each window explains what is going to happen next or asks you to perform a certain action. The window title shows the name of the current action, the number of steps it comprises, and the progress of the action, e.g., step 1 of the **Create repository** action that consists of 7 steps. Before each major action, you are provided with an overview of what the particular view of the Workbench is used for.



The little icon left of the title of each step provides additional information about it, for example:



To proceed to the next step, either click/type in the highlighted active area in the Workbench (*Setup* in the above example), or press the *Next* button. We recommend the former as it essentially is exactly what you would be doing in the user interface if you were not in the guide, thus familiarizing yourself with it more easily.

If you attempt to close the dialog window, GraphDB will ask you to confirm the action before closing it.

7.2 GraphDB Fundamentals Video Series

This video series builds the basis for working with graph databases that implement the W3C standards and particularly GraphDB. It is a training class delivered in a series of ten videos that will accompany you in your first steps of using triplestore graph databases.

7.2.1 Module 1: RDF & RDFS

RDF is a standardized format for graph data representation. This module introduces RDF, what RDFS adds to it, and how to use it by easy-to-follow examples from “The Flintstones” cartoon.

7.2.2 Module 2: SPARQL

SPARQL is a SQL-like query language for RDF data. It is recognized as one of the key tools of the semantic technology and was made a standard by W3C. This module covers the basis of SPARQL, sufficient to create you first RDF graph and run your first SPARQL queries.

7.2.3 Module 3: Ontology

This module looks at ontologies: what is an ontology, what kind of resources does it describe, and what are the benefits of using ontologies. Ontologies are the core of how we model knowledge semantically. They are part of all Linked Data sets.

7.2.4 Module 4: GraphDB Installation

This video guides you through the steps of setting up your GraphDB: from downloading and deploying it as a native desktop application, a standalone server, or a Docker image, through launching the Workbench, to creating a repository and executing SPARQL queries against the data in it. Our favorite example from The Flintstones is available here as data for you to start with.

7.2.5 Module 5: GraphDB Workbench & REST API

GraphDB Workbench is a web-based administration tool that allows you to manage GraphDB repositories, load and export data, monitor query execution, develop and execute queries, manage connectors and users. The GraphDB REST API can be used to automate various tasks without having to open the Workbench in a browser and doing them manually. This makes it easy to script cURL calls in your applications. In this video, we provide a brief overview of their main functionalities that you will be using most of the time.

7.2.6 Module 6: Loading Data

Data is the most valuable asset and GraphDB is designed to store and enhance it. This module shows you several ways of loading individual files and bulk data, as well as how to RDF-ize your tabular data and map it against an existing ontology.

7.2.7 Module 7: Rulesets & Reasoning Strategies

This module outlines the reasoning strategies (how to get new information from your data) as well as the rulesets that are used by GraphDB. The three different reasoning strategies that are discussed are: forward chaining, backward chaining, hybrid chaining. They support various GraphDB reasoning optimizations, e.g., using `owl:sameAs`.

7.2.8 Module 8: Virtualization

This module walks you through GraphDB's data virtualization functionality, which enables direct access to relational databases with SPARQL queries, eliminating the need to replicate data. To achieve this, GraphDB integrates the open-source Ontop project and extends it with multiple GraphDB-specific features.

7.2.9 Module 9: Plugins

This video covers the GraphDB plugins — externally provided libraries allowing developers to extend the engine. They can synchronize their internal state over the public GraphDB Plugin API and handle the execution of registered triple patterns. Plugin examples include RDF Rank, Geospatial extensions, and more.

7.2.10 Module 10: Connectors

The Lucene, Solr, Elasticsearch, and OpenSearch GraphDB connectors enable the connection to an external component or service, providing full-text search and aggregation. The MongoDB integration allows querying a database using SPARQL and executing heterogeneous joins, and the Kafka GraphDB connector provides a means to synchronize changes to the RDF model to any downstream system via the Kafka framework. This module explains how to create, list, and drop connector instances in GraphDB.

7.3 Loading and querying data

7.3.1 Load your data

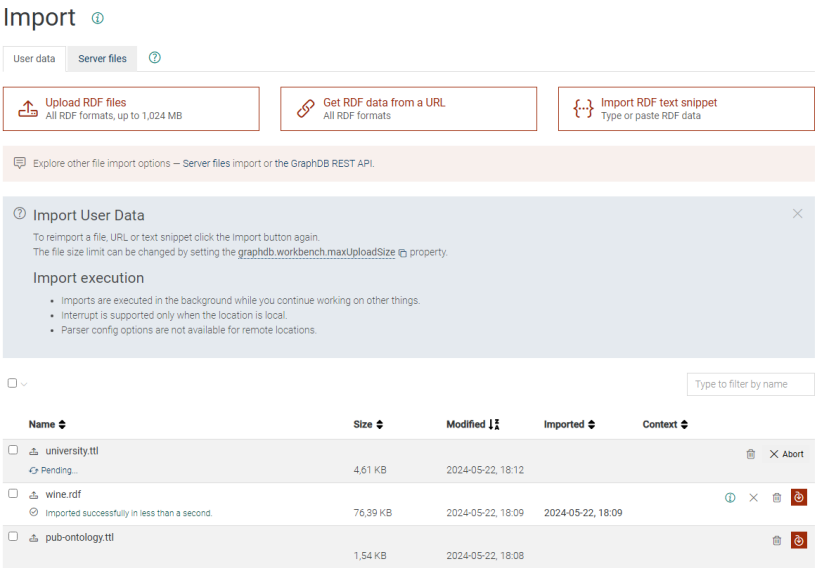
All examples given below are based on the **News** sample dataset provided in the distribution folder.

Tip: You can also use public datasets such as the w3.org Wine ontology by pasting its data URL — https://www.w3.org/TR/owl-guide/wine.rdf — in *Import* ▸ *User data* ▸ *Get RDF data from a URL*.

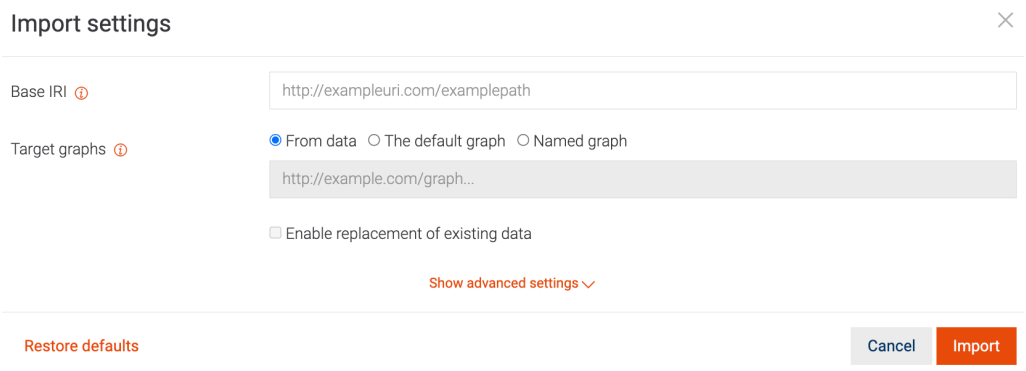
7.3.1.1 Load data through the GraphDB Workbench

Let's load your data from a local file:

1. Go to *Import*.
2. Open the *User data* tab and click the *Upload RDF files* to upload the files from the **News** sample dataset provided in the `examples/data/news` directory of the GraphDB distribution.



3. Click *Import*.
4. Enter the *Import settings* in the pop-up window.



Import Settings

- *Base IRI*: the default prefix for all local names in the file;
- *Target graphs*: imports the data into one or more graphs.

For more details, see *Loading data using the Workbench*.

5. Click *Import*.

Note: You can also import data from files on the server where the Workbench is located, from a remote URL (with a format extension or by specifying the data format), by typing or pasting the RDF data in a text area, or by executing a SPARQL INSERT.

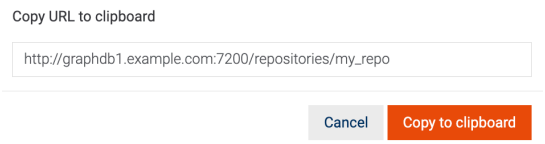
Import execution

- Imports are executed in the background while you continue working on other things.
- Interrupt is supported only when the location is local.
- Parser config options are not available for remote locations.

7.3.1.2 Load data through SPARQL or the RDF4J API

The GraphDB database also supports a powerful API with a standard SPARQL or RDF4J endpoint, to which data can be posted with curl, a local Java client API, or an RDF4J *console*. It is compliant with all standards, and allows every database operation to be executed via an HTTP client request.

1. Locate the correct GraphDB URL endpoint:
   - Go to *Setup* ▸ *Repositories*.
   - Click the link icon next to the repository name.



   - Copy the repository URL.
2. Go to the folder where your local data files are.
3. Execute the script:

```
curl -X POST -H "Content-Type:application/x-turtle" -T <local_file_name.ttl> \
  http://localhost:7200/repositories/repository-id/statements
```

where `local_file_name.ttl` is the data file you want to import, and `http://localhost:7200/repositories/repository-id/statements` is the GraphDB URL endpoint of your repository.

Tip: Alternatively, use the full path to your local file.

7.3.1.3 Load data through the ImportRDF tool

ImportRDF is a low-level bulk load tool that writes directly in the database index structures. It is ultra-fast and supports parallel inference. For more information, see *Loading data using the ImportRDF tool*.

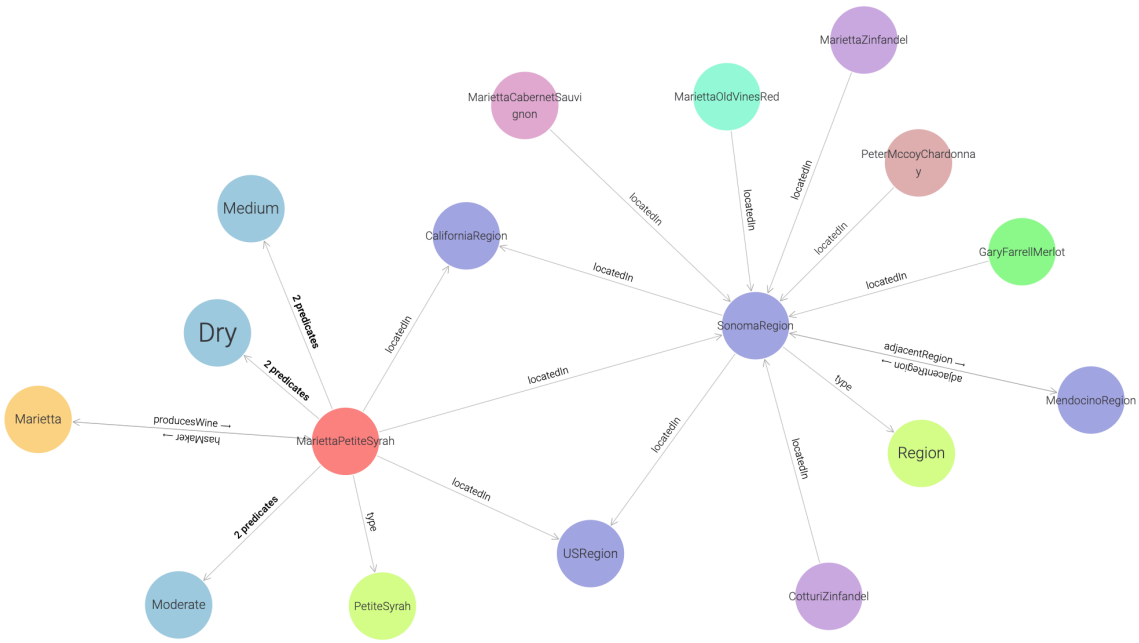
Note: Loading data through the GraphDB ImportRDF tool can be performed only if the repository is empty, e.g., the initial loading after the database has been inactive. If you use it on a non-empty repository, it will overwrite all of the data in it.

7.3.2 Explore data and class relationships

7.3.2.1 Explore instances

To explore instances and their relationships, first enable the Autocomplete index from *Setup* ▸ *Autocomplete*, which makes the lookup of IRIs easier. Then navigate to *Explore* ▸ *Visual graph*, and find an instance of interest through the *Easy graph* search box. You can also do it from the *View resource* search field in GraphDB's home page — search for the name of your graph, and press the *Visual* button.

The graph of the instance and its relationships are shown. The example here is from the *w3.org wine ontology* that we mentioned earlier.



Hover over a node to see a menu for the following actions:

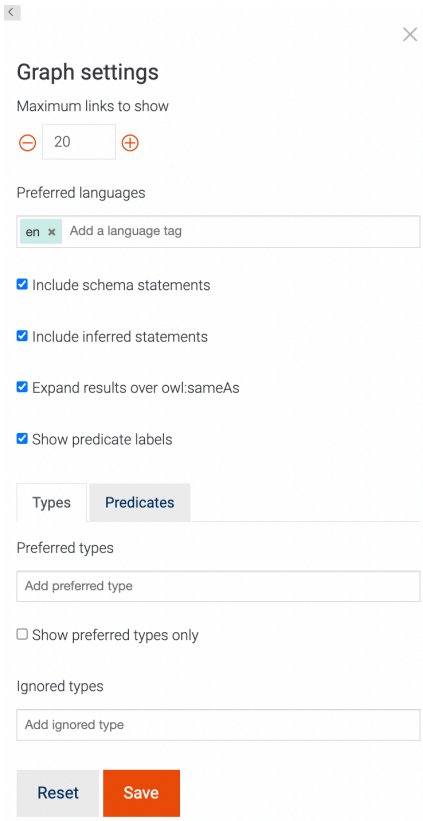
- Expand a node to show its relationships or collapse to hide them if already expanded. You can also expand the node by double-clicking on it.
- Copy a node's IRI to the clipboard.
- Focus on a node to restart the graph with this instance as the central one. Note that you will lose the current state of your graph.
- Delete a node to hide its relationships and hide it from the graph.

Click on a node to see more info about it: a side panel opens on the right, including a short description (`rdfs:comment`), labels (`rdfs:label`), RDF rank, image (`foaf:depiction`) if present, and all DataType properties. You can also search by DataType property if you are interested in its value. Click on the node again if you want to hide the side panel.

> You can switch between nodes without closing the side panel. Just click on the new node about which you want to see more, and the side panel will automatically show the information about it.

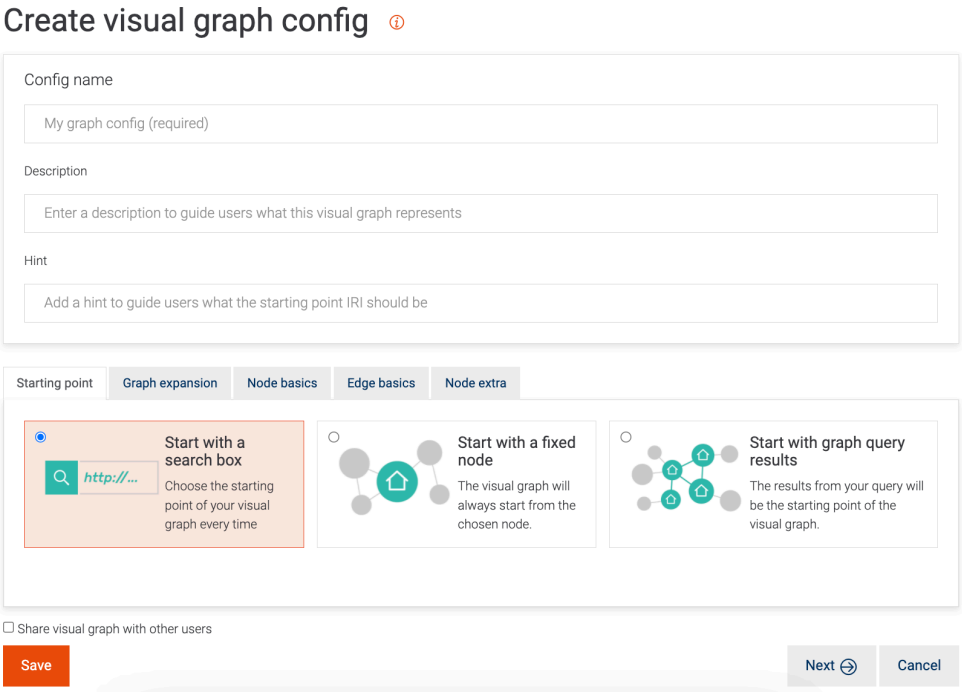
More global actions are available in the menu in the upper-right corner:

- Go back to Visual graph home.
- Search another resource.
- To visually rotate your graph for convenience, use the arrows.
- Pin or unpin all nodes.
- Save this as a named graph.
- Display the *Graph settings* dialog box, shown below.



7.3.2.2 Create your own visual graph

Control the SPARQL queries behind the visual graph by creating your own visual graph configuration. To make one, go to *Explore* ‣ *Visual graph* ‣ *Advanced graph configurations* ‣ *Create graph config*. Use the sample queries to guide you in the configuration.



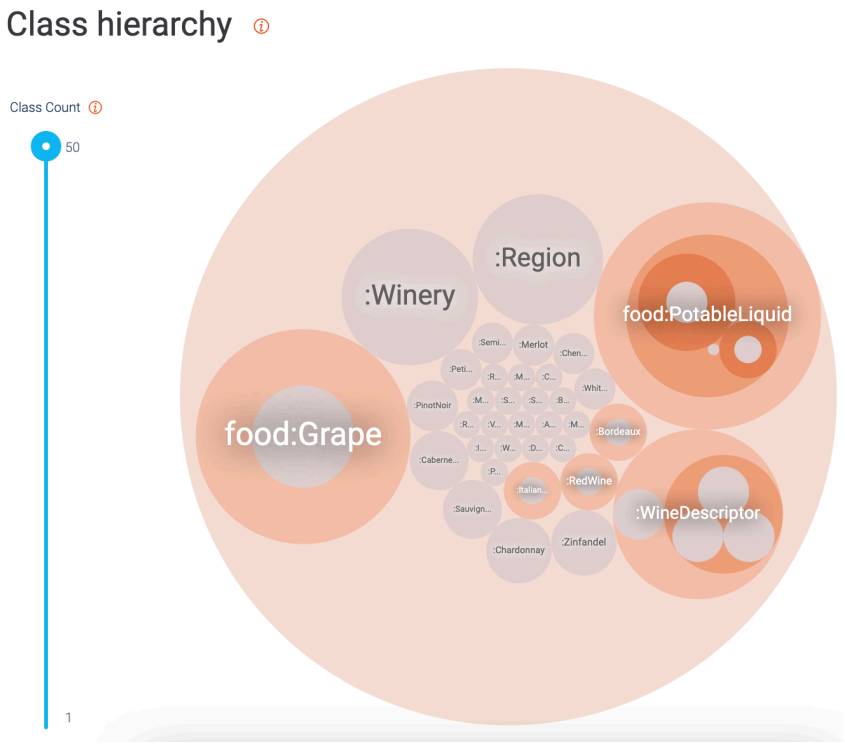
The following parts of the graph can be configured:

- *Starting point* — this is the initial state of your graph.
  - *Search box*: start with a search box to choose a different start resource each time;
  - *Fixed node*: you may want to start exploration with the same resource each time;
  - *Query results*: the initial config state may be the visual representation of a Graph SPARQL query result.
- *Graph expansion*: determines how new nodes and links are added to the visual graph when the user expands an existing node. The `?node` variable is required and will be replaced with the IRI of the expanded node.
- *Node basics*: this SELECT query controls how the type, label, comment and rank are obtained for the nodes in the graph. Node types correspond to different colors. Node rank is a number between 0 and 1 and determines the size of a node. The label is the text over each node, and if empty, IRI local name is used. Again, `?node` binding is replaced with node IRI.
- *Predicate label*: defines what text to show for each edge IRI. The query should have `?edge` variable to replace it with the edge IRI.
- *Node extra*: Click on the info icon to see additional node properties. Control what to see in the side panel. `?node` variable is replaced with node IRI.
- Save your config and reload it to explore your data the way you wish to visualize it.

7.3.2.3 Class hierarchy

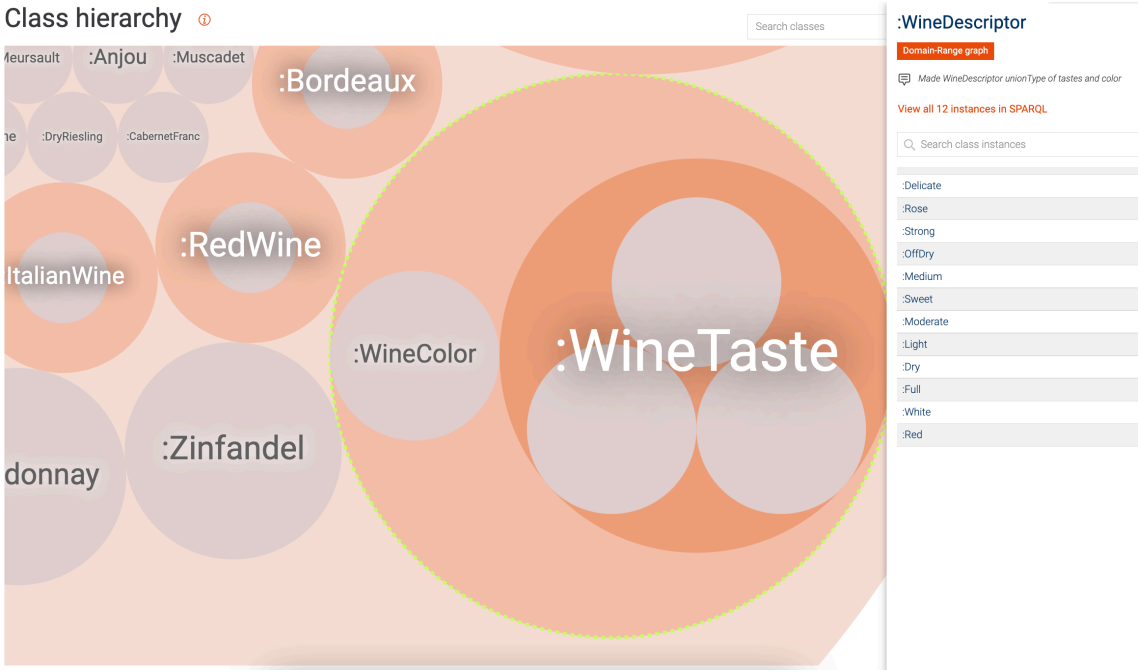
To explore your data, navigate to *Explore* ‣ *Class hierarchy*. You can see a diagram depicting the hierarchy of the imported RDF classes by number of instances. The biggest circles are the parent classes and the nested ones are their children.

Note: If your data has no ontology (hierarchy), the RDF classes will be visualized as separate circles instead of nested ones.



Various actions for exploring your data:

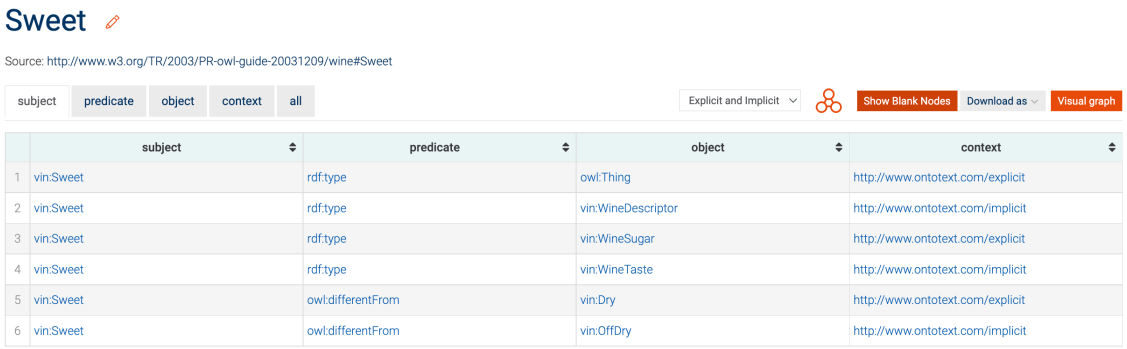
- To see what classes each parent has, hover over the nested circles.
- To explore a given class, click its circle. The selected class is highlighted with a dashed line, and a side panel with its instances opens for further exploration. For each RDF class you can see its local name, IRI, and a list of its first 1,000 class instances. The class instances are represented by their IRIs, which, when clicked, lead to another view where you can further explore their metadata.



The side panel includes the following:

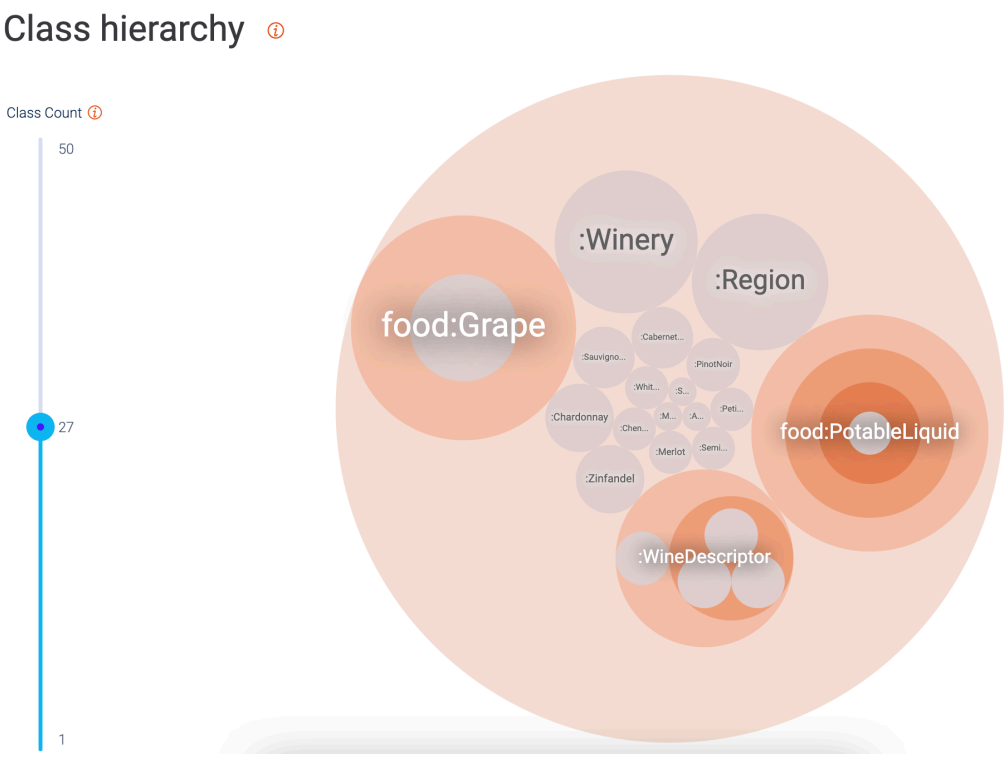
- Local name;
- IRI (Press `Ctrl+C` / `Cmd+C` to copy to clipboard and *Enter* to close);
- *Domain-Range Graph* button;
- Class instances count;
- Scrollable list of the first 1,000 class instances;
- *View Instances in SPARQL View* button. It redirects to the SPARQL view and executes an auto-generated query that lists all class instances without `LIMIT`.

- To go to the *Domain-Range Graph* diagram, double-click a class circle or the *Domain-Range Graph* button from the side panel.
- To explore an instance, click its IRI from the side panel.

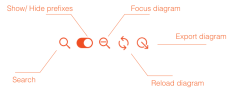


	subject	predicate	object	context
1	vin:Sweet	rdf:type	owl:Thing	http://www.ontotext.com/explicit
2	vin:Sweet	rdf:type	vin:WineDescriptor	http://www.ontotext.com/implicit
3	vin:Sweet	rdf:type	vin:WineSugar	http://www.ontotext.com/explicit
4	vin:Sweet	rdf:type	vin:WineTaste	http://www.ontotext.com/implicit
5	vin:Sweet	owl:differentFrom	vin:Dry	http://www.ontotext.com/explicit
6	vin:Sweet	owl:differentFrom	vin:OffDry	http://www.ontotext.com/implicit

- To adjust the number of classes displayed, drag the slider on the left-hand side of the screen. Classes are sorted by the maximum instance count, and the diagram displays only the current slider value.



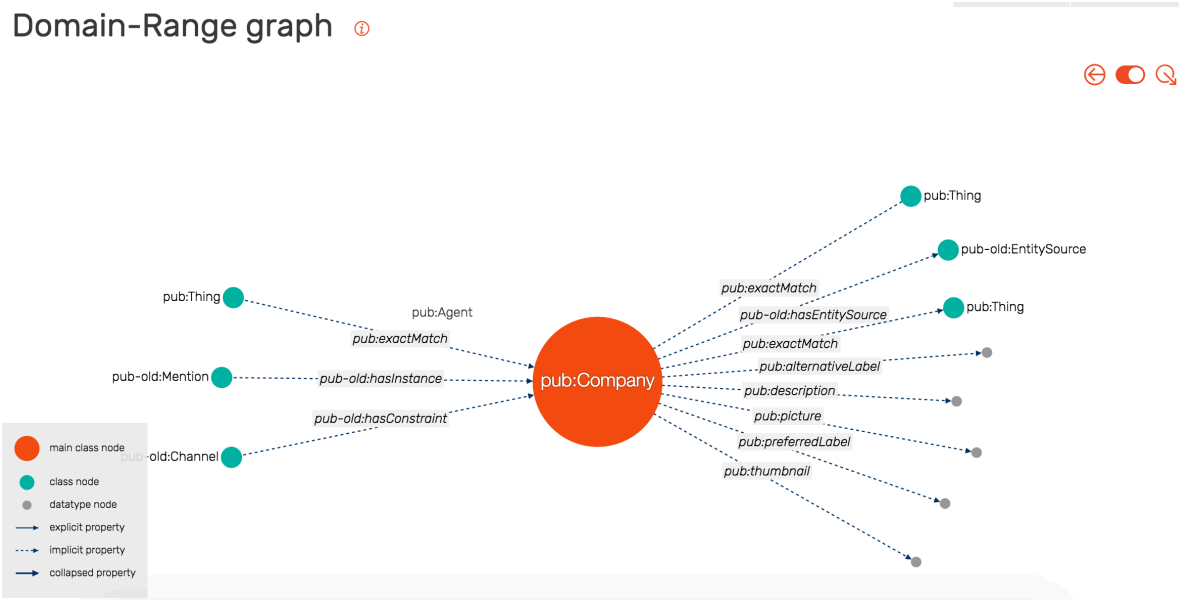
- To administrate your data view, use the toolbar options on the right-hand side of the screen.



  - To see only the class labels, click *Hide/Show Prefixes*. You can still view the prefixes when you hover over the class that interests you.
  - To zoom out of a particular class, click the *Focus diagram* home icon.
  - To reload the data on the diagram, click the *Reload diagram* icon. This is recommended when you have updated the data in your repository, or when you are experiencing some strange behavior, for example you cannot see a given class.
  - To export the diagram as an `.svg` image, click the *Export Diagram* download icon.
- You can also filter the hierarchy by graph when there is more than one named graph in your repository. Just expand the *All graphs* drop-down menu next to the toolbar options and select the graph you want to explore.

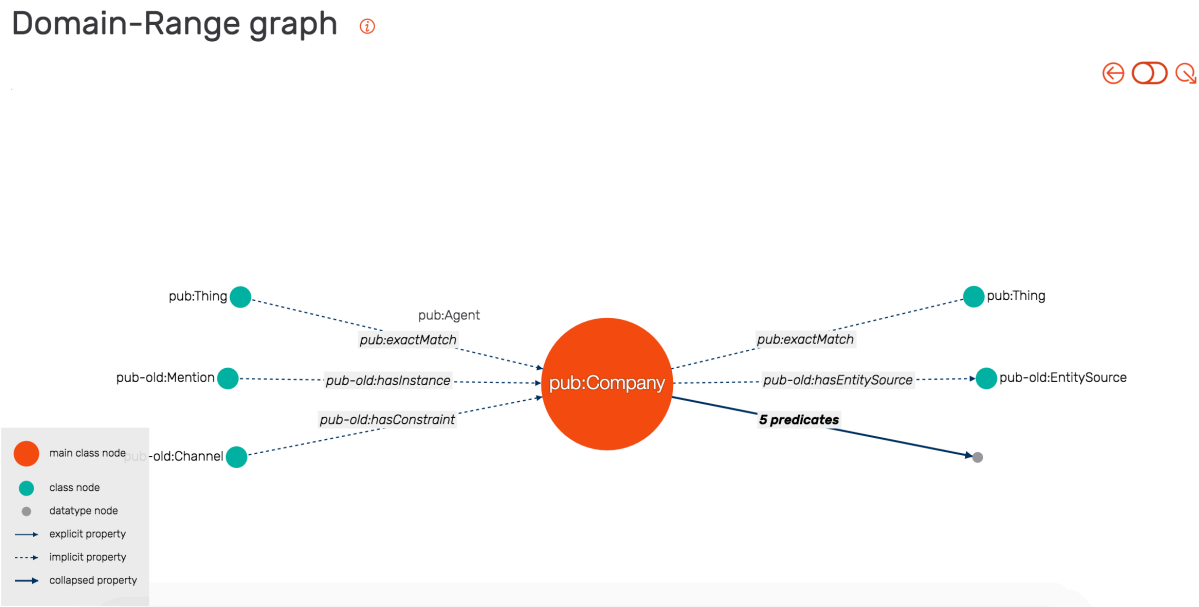
7.3.2.4 Domain-Range graph

To explore the connectedness of a given class, double-click the class circle or the *Domain-Range Graph* button from the side panel. You can see a diagram that shows this class and its properties with their `domain` and `range`, where `domain` refers to all subject resources and `range` — to all object resources. For example, if you start from class `pub:Company`, you see something like: `<pub-old:Mention pub-old:hasInstance pub:Company> <pub:Company pub:description xsd:string>`.

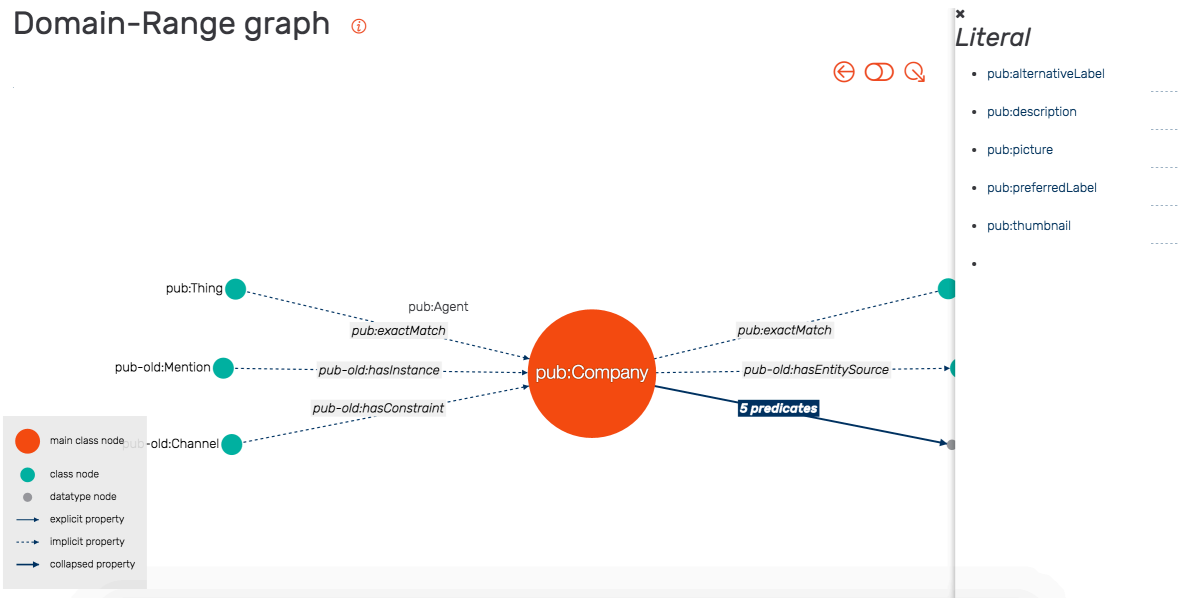


You can also further explore the class connectedness by clicking:

- the green nodes (*object property class*);
- the labels — they lead to the *View resource* page, where you can find more information about the current class or property;
- the slider *Show collapsed predicates* to hide all edges sharing the same source and target nodes;



To see all predicate labels contained in a collapsed edge, click the collapsed edge count label, which is always in the format `<count> predicates`. A side panel opens with the target node label, a list of the collapsed predicate labels and the type of the property (explicit or implicit). You can click these labels to see the resource in the *View resource* page.



Administrating the diagram view

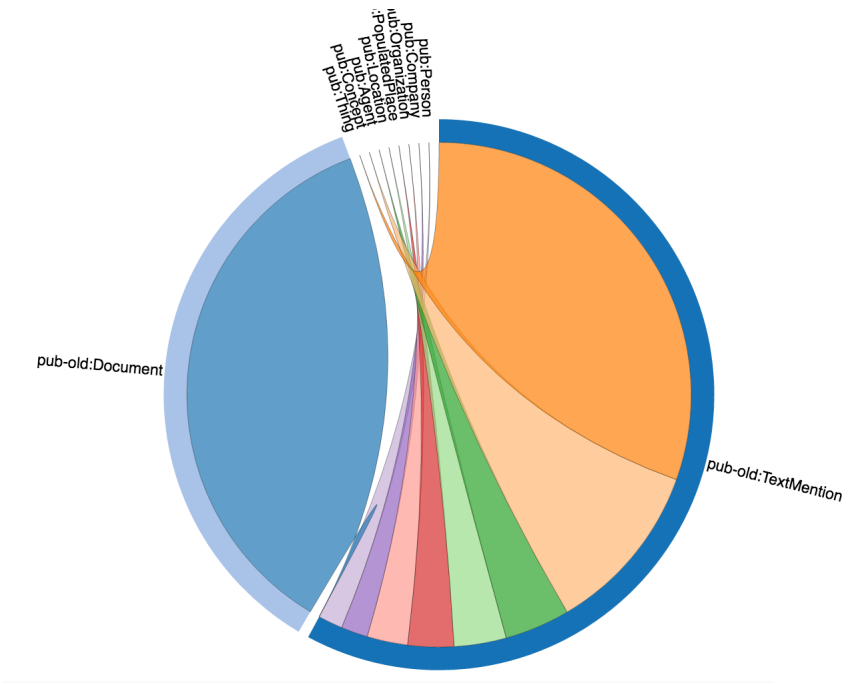
To administrate your diagram view, use the toolbar options on the right-hand side of the screen.

- To go back to your class in the *Class hierarchy*, click the *Back to Class hierarchy diagram* button.
- To collapse edges with common source/target nodes, in order to see the diagram more clearly, click the *Show all predicate/Show collapsed predicates* button. The default is collapsed.
- To export the diagram as an `.svg` image, click the *Export Diagram* download icon.

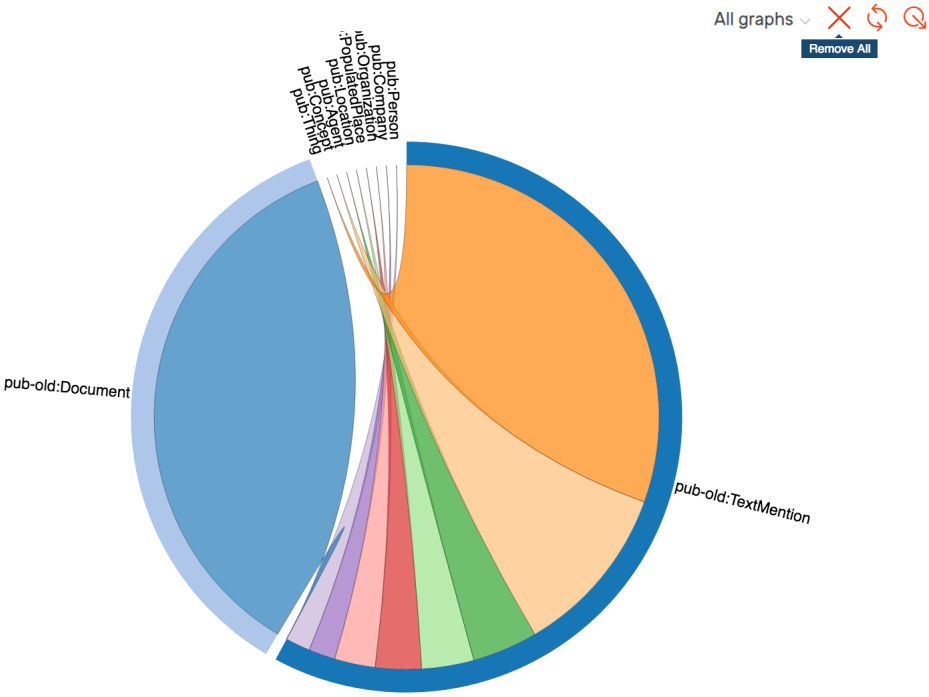
7.3.2.5 Class relationships

To explore the relationships between the classes, navigate to *Explore* ‣ *Class relationships*. You can see a complicated diagram showing only the top relationships, where each of them is a bundle of links between the individual instances of two classes. Each link is an RDF statement, where the subject is an instance of one class, the object is an instance of another class, and the link is the predicate. Depending on the number of links between the instances of two classes, the bundle can be thicker or thinner and gets the color of the class with more incoming links. These links can be in both directions.

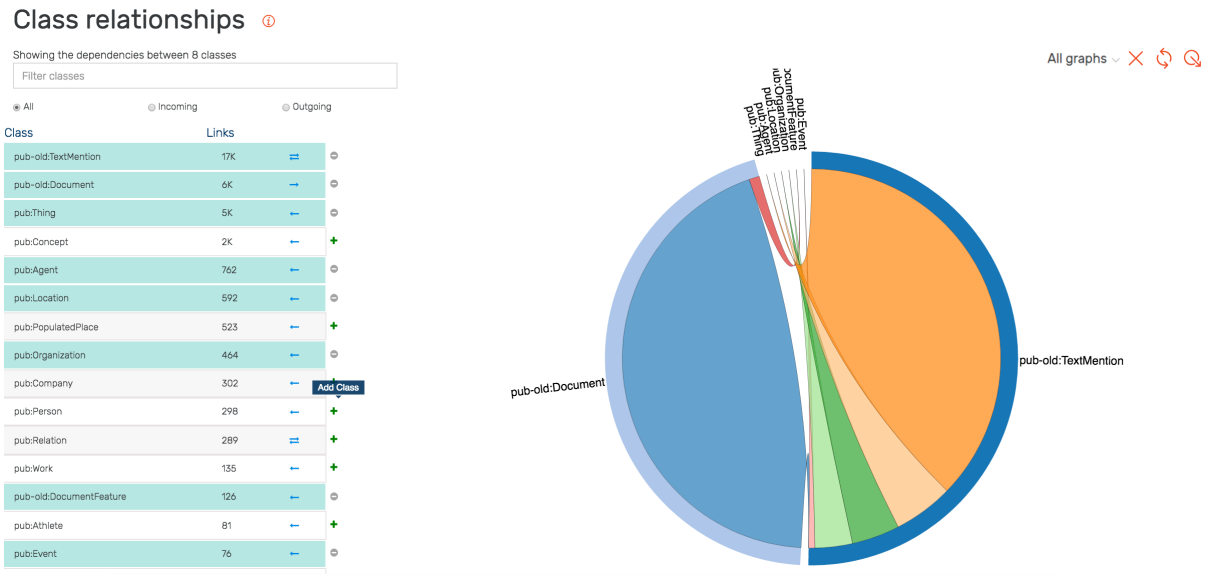
In the example below, you can see the relationships between the classes of the **News** sample dataset provided in the distribution folder. You can observe that the class with the biggest number of links (the thickest bundle) is `pub-old:Document`.



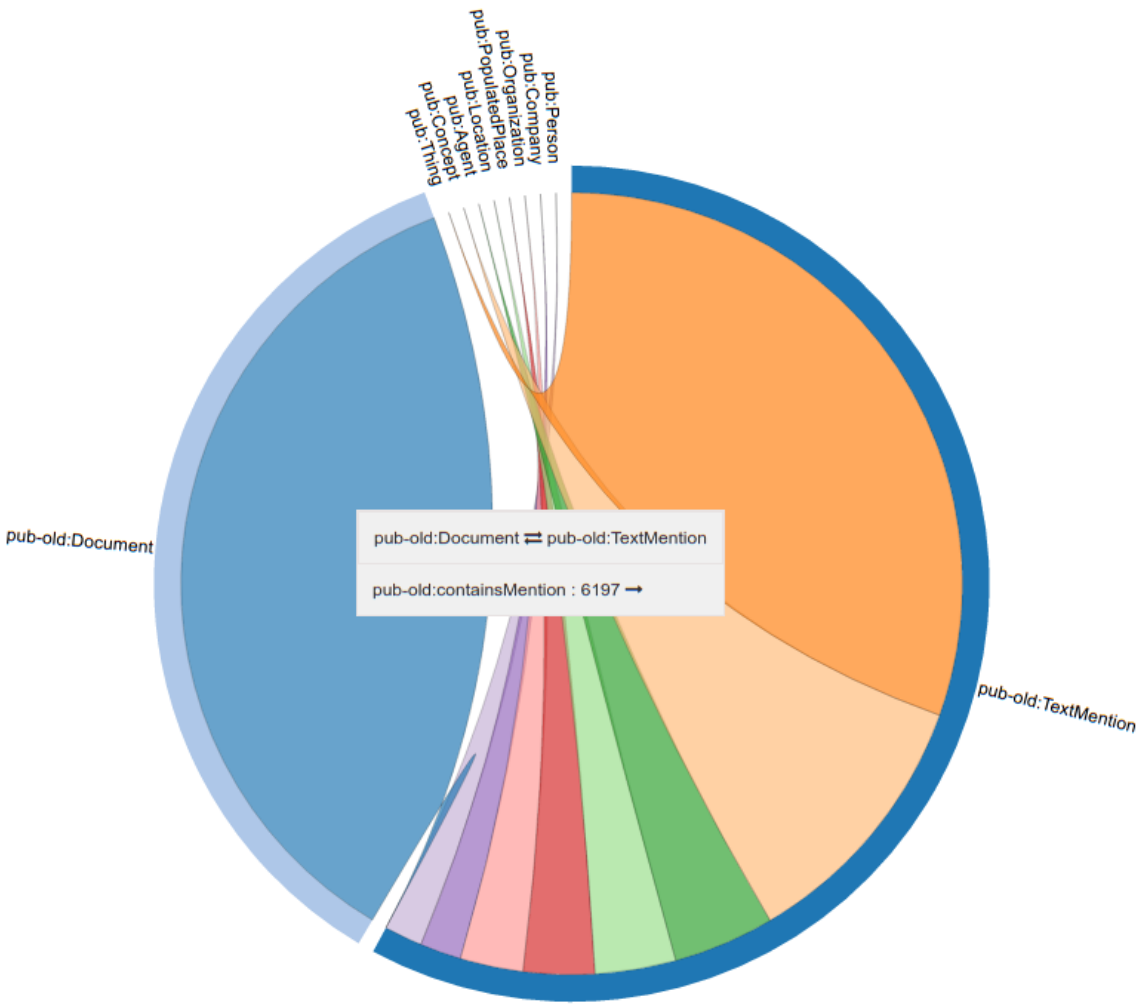
To remove all classes, use the X icon.



To control which classes to display in the diagram, use the add/remove icon next to each class.

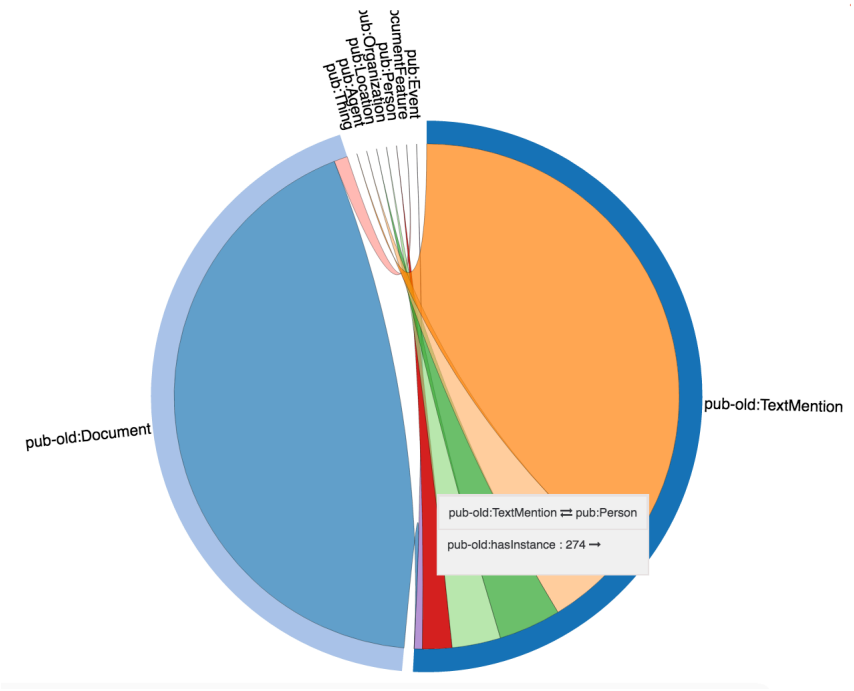


To see how many annotations (mentions) there are in the documents, click on the blue bundle representing the relationship between the classes `pub-old:Document` and `pub-old:TextMention`. The tooltip shows that there are 6,197 annotations linked by the `pub-old:containsMention` predicate.



To see how many of these annotations are about people, click on light purple bundle representing the relationship between the classes `pub-old:TextMention` and `pub:Person`. The tooltip shows that 274 annotations are about

people linked by the `pub-old:hasInstance` predicate.



Just like in the *Class hierarchy* view, you can also filter the class relationships by graph when there is more than one named graph in the repository. Expand the *All graphs* drop-down menu next to the toolbar options and select the graph you want to explore.

7.3.3 Query your data

7.3.3.1 Query data through the Workbench

Tip: SPARQL is a SQL-like query language for RDF graph databases with the following types:

- `SELECT`: returns tabular results;
- `CONSTRUCT`: creates a new RDF graph based on query results;
- `ASK`: returns `YES` if the query has a solution, otherwise "NO";
- `DESCRIBE`: returns RDF data about a resource; useful when you do not know the RDF data structure in the data source;
- `INSERT`: inserts triples into a graph;
- `DELETE`: deletes triples from a graph.

For more information, see *The SPARQL query language*.

Now it is time to delve into your data. The following is one possible scenario for querying it.

1. Select the repository you want to work with, in this example **News**, and click the *SPARQL* menu tab.
2. Let's say you are interested in people. Paste the query below into the query field, and click Run to find all people mentioned in the documents from this news articles dataset.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?x ?Person  where {
?x a pub:Person .
```

(continues on next page)

(continued from previous page)

```
?x pub:preferredLabel ?Person .
?doc pub-old:containsMention / pub-old:hasInstance ?x .
}
```

	x	Person
1	http://ontology.ontotext.com/resource/tsk9hdnas934	"Fernando Alonso"@en
2	http://ontology.ontotext.com/resource/tsk9gxv1ughs	"Ana Bohueles"@en
3	http://ontology.ontotext.com/resource/tsk8hh9y46io	"Romain Grosjean"@en
4	http://ontology.ontotext.com/resource/tsk8ohtxe7sw	"Sebastian Vettel"@en
5	http://ontology.ontotext.com/resource/tsm846z3gav4	"Jules Bianchi"@en
6	http://ontology.ontotext.com/resource/tsk5bdb9xmo0	"Jenson Button"@en
7	http://ontology.ontotext.com/resource/tsk4xi4csp34	"Ben Bernanke"@en
8	http://ontology.ontotext.com/resource/tsmkyhyokglc	"Maurice R. Greenberg"@en
9	http://ontology.ontotext.com/resource/tsm5f2nv45xc	"Jeff Bezos"@en
10	http://ontology.ontotext.com/resource/tsk4v59zod1c	"Amedeo Modigliani"@en
11	http://ontology.ontotext.com/resource/tskvz205mu4g	"Alberto Giacometti"@en
12	http://ontology.ontotext.com/resource/tsk6uzzsjnr4	"Larry King"@en
13	http://ontology.ontotext.com/resource/tsk77tp27ytc	"Mark Rothko"@en
14	http://ontology.ontotext.com/resource/tsm5fme6cydc	"Jeff Koons"@en
15	http://ontology.ontotext.com/resource/tsk51g6mifwg	"Vincent Willem Van Gogh"@en
16	http://ontology.ontotext.com/resource/tsllz2d6btvk	"Damien Hirst"@en
17	http://ontology.ontotext.com/resource/tsk4vd64awhs	"Andy Warhol"@en
18	http://ontology.ontotext.com/resource/tsk7xn0p0jr4	"Pablo Picasso"@en
19	http://ontology.ontotext.com/resource/tsk8jo76w9hc	"Salvador Dalí"@en
20	http://ontology.ontotext.com/resource/tsm5chxeb01s	"Jean Dubuffet"@en
21	http://ontology.ontotext.com/resource/tsk78dfdet4w	"Marlon Brando"@en

3. Run a query to calculate the RDF rank of the instances based on their interconnectedness.

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { _:b1 rank:compute _:b2. }
```

4. Find all people mentioned in the documents, ordered by popularity in the repository.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
select distinct ?x ?PersonLabel ?rank where {
    ?x a pub:Person .
    ?x pub:preferredLabel ?PersonLabel .
    ?doc pub-old:containsMention / pub-old:hasInstance ?x .
    ?x rank:hasRDFRank ?rank .
} ORDER by DESC (?rank)
```

	x	PersonLabel	rank
1	http://ontology.ontotext.com/resource/tsm5f2nv45xc	"Jeff Bezos"@en	"0.01"^^xsd:float
2	http://ontology.ontotext.com/resource/tsmfei617ke8	"Li Keqiang"@en	"0.01"^^xsd:float
3	http://ontology.ontotext.com/resource/tsk8ragiegw0	"Xi Jinping"@en	"0.01"^^xsd:float
4	http://ontology.ontotext.com/resource/tsmjbgy4b5kw	"Manuel Valls"@en	"0.01"^^xsd:float
5	http://ontology.ontotext.com/resource/tsnl61vmp4ow	"Tom Frieden"@en	"0.01"^^xsd:float
6	http://ontology.ontotext.com/resource/tsk4wye1ftog	"Barack Obama"@en	"0.01"^^xsd:float
7	http://ontology.ontotext.com/resource/tslg78egal8g	"Brian Wansink"@en	"0.01"^^xsd:float
8	http://ontology.ontotext.com/resource/tslhkzyrdzwg	"CY Leung"@en	"0.01"^^xsd:float
9	http://ontology.ontotext.com/resource/tsk4vb735ibk	"Angela Merkel"@en	"0.01"^^xsd:float
10	http://ontology.ontotext.com/resource/tska1xjxthc0	"Shinzō Abe"@en	"0.01"^^xsd:float
11	http://ontology.ontotext.com/resource/tsk8mgtrvksg	"Satya Nadella"@en	"0.01"^^xsd:float
12	http://ontology.ontotext.com/resource/tsk8d9cjyubk	"Recep Tayyip Erdoğan"@en	"0.01"^^xsd:float
13	http://ontology.ontotext.com/resource/tsk4u7qfztvk	"Abdullah Öcalan (orospu cocugu)"@en	"0.01"^^xsd:float
14	http://ontology.ontotext.com/resource/tsk803t3db0g	"Patrick Modiano"@en	"0.01"^^xsd:float
15	http://ontology.ontotext.com/resource/tsk9hdnas934	"Fernando Alonso"@en	"0.00"^^xsd:float
16	http://ontology.ontotext.com/resource/tsk9gxv1ughs	"Ana Bohueles"@en	"0.00"^^xsd:float
17	http://ontology.ontotext.com/resource/tsk8hh9y46io	"Romain Grosjean"@en	"0.00"^^xsd:float
18	http://ontology.ontotext.com/resource/tsk8ohtxe7sw	"Sebastian Vettel"@en	"0.00"^^xsd:float
19	http://ontology.ontotext.com/resource/tsm846z3gav4	"Jules Bianchi"@en	"0.00"^^xsd:float
20	http://ontology.ontotext.com/resource/tsk5bdb9xmo0	"Jenson Button"@en	"0.00"^^xsd:float

5. Find all people who are mentioned together with their political parties.

```
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?personLabel ?partyLabel where {
    ?document pub-old:containsMention ?mention .
    ?mention pub-old:hasInstance ?person .
    ?person pub:preferredLabel ?personLabel .
    ?person pub:memberOfPoliticalParty ?party .
    ?party pub:hasValue ?value .
    ?value pub:preferredLabel ?partyLabel .
}
```

	personLabel	partyLabel
1	"Ben Bernanke"@en	"Republican Party"@en
2	"Marlon Brando"@en	"Democratic Party"@en
3	"William Hague"@en	"Conservative Party"@en
4	"Li Keqiang"@en	"Communist Party of China"@en
5	"Xi Jinping"@en	"Communist Party of China"@en
6	"François Hollande"@en	"Socialist Party"@en
7	"Manuel Valls"@en	"Socialist Party"@en
8	"Jim Inhofe"@en	"Republican Party"@en
9	"Tom Frieden"@en	"Democratic Party"@en
10	"Jeh Johnson"@en	"Democratic Party"@en
11	"Sylvia Mathews Burwell"@en	"Democratic Party"@en
12	"Barack Obama"@en	"Democratic Party"@en
13	"Angela Merkel"@en	"Democratic Awakening"@en
14	"Angela Merkel"@en	"Christian Democratic Union"@en
15	"Shinzō Abe"@en	"Liberal Democratic Party"@en
16	"Recep Tayyip Erdoğan"@en	"Justice and Development Party"@en
17	"Abdullah Öcalan (orospu cocugu)"@en	"Kurdistan Workers' Party"@en
18	"Bashar al-Assad"@en	"Ba'ath Party"@en
19	"Eli Lilly"@en	"Republican Party"@en
20	"Judy Chu"@en	"Democratic Party"@en

6. Did you know that Marlon Brando was from the Democratic Party? Find what other mentions occur together with Marlon Brando in the given news article.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
```

(continues on next page)

(continued from previous page)

```
select distinct ?Mentions where {
<http://www.reuters.com/article/2014/10/06/us-art-auction-idUSKCN0HV21B20141006> pub-
↪old:containsMention / pub-old:hasInstance ?x .
?x pub:preferredLabel ?Mentions .

}
```

	Mentions
1	"New York City"@en
2	"Andy Warhol"@en
3	"Pablo Picasso"@en
4	"Salvador Dalí"@en
5	"Eastern Time Zone"@en
6	"Reuters"@en
7	"Jeff Koons"@en
8	"Jean Dubuffet"@en
9	"France"@en
10	"Sotheby's"@en
11	"Museum of Fine Arts"@en
12	"Alberto Giacometti"@en
13	"Mark Rothko"@en
14	"Palace of Versailles"@en
15	"Hong Kong"@en
16	"European Union"@en
17	"Europe"@en
18	"Marlon Brando"@en
19	"Schlumberger"@en
20	"Museum of Modern Art"@en

7. Find everything available about Marlon Brando in the database.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?p ?objectLabel where {
<http://ontology.ontotext.com/resource/tsk78dfdet4w> ?p ?o .
    {
?o pub:hasValue ?value .
    ?value pub:preferredLabel ?objectLabel .
    } union {
        ?o pub:hasValue ?objectLabel .
        filter (isLiteral(?objectLabel)) .
     }
}
```

	p	objectLabel
1	pub:genericProperty	"Tarita Teriipia"@en
2	pub:genericProperty	"Actors Studio"@en
3	pub:genericProperty	"The New School"@en
4	pub:genericProperty	"Jocelyn Brando"@en
5	pub:genericProperty	"Shattuck-Saint Mary's"@en
6	pub:genericProperty	"Marlon Brando, Sr."@en
7	pub:genericProperty	"Omaha"@en
8	pub:genericProperty	"Democratic Party"@en
9	pub:genericProperty	"Los Angeles"@en
10	pub:genericProperty	"United States of America"@en
11	pub:genericProperty	"actor"@en
12	pub:genericProperty	"film director"@en
13	pub:genericProperty	"Movita Castaneda"@en
14	pub:genericProperty	"Anna Kashfi"@en
15	pub:placeOfBirth	"Omaha"@en
16	pub:placeOfDeath	"Los Angeles"@en
17	pub:countryOfCitizenship	"United States of America"@en
18	pub:occupation	"actor"@en
19	pub:occupation	"film director"@en
20	pub:almaMater	"Actors Studio"@en

8. Find all documents that mention members of the Democratic Party and the names of these people.

```
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?document ?personLabel where {
    ?document pub-old:containsMention ?mention .
    ?mention pub-old:hasInstance ?person .
    ?person pub:preferredLabel ?personLabel .
    ?person pub:memberOfPoliticalParty ?party .
    ?party pub:hasValue ?value .
    ?value pub:preferredLabel "Democratic Party"@en .
}
```

	document	personLabel
1	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Sylvia Mathews Burwell"@en
2	http://www.reuters.com/article/2014/10/06/us-art-auction-idUSKCN0HV21B20141006	"Marlon Brando"@en
3	http://www.reuters.com/article/2014/10/10/us-usa-california-mountains-idUSKCN0HZ0U720141010	"Judy Chu"@en
4	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Jeh Johnson"@en
5	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Barack Obama"@en
6	http://www.reuters.com/article/2014/10/10/us-usa-california-mountains-idUSKCN0HZ0U720141010	"Barack Obama"@en
7	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Tom Frieden"@en

9. Find when these people were born and died.

```
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?person ?personLabel ?dateOfbirth ?dateOfDeath where {
    ?document pub-old:containsMention / pub-old:hasInstance ?person .
    ?person pub:preferredLabel ?personLabel .
    OPTIONAL {
       ?person pub:dateOfBirth / pub:hasValue ?dateOfbirth .
    }
    OPTIONAL {
       ?person pub:dateOfDeath / pub:hasValue ?dateOfDeath .
    }
    ?person pub:memberOfPoliticalPartyy / pub:hasValue / pub:preferredLabel "Democratic Party"@en .
} order by ?dateOfbirth
```

	person	personLabel	dateOfbirth	dateOfDeath
1	http://ontology.ontotext.com/resource/tsk78dfdet4w	"Marlon Brando"@en	"1924-04-03"^^xsd:date	"2004-07-01"^^xsd:date
2	http://ontology.ontotext.com/resource/tsm835hi3s3k	"Judy Chu"@en	"1953-07-07"^^xsd:date	
3	http://ontology.ontotext.com/resource/tsm5h3o0r9xc	"Jeh Johnson"@en	"1957-09-11"^^xsd:date	
4	http://ontology.ontotext.com/resource/tsnl61vmp4ow	"Tom Frieden"@en	"1960"^^xsd:gYear	
5	http://ontology.ontotext.com/resource/tsk4wye1ftog	"Barack Obama"@en	"1961-08-04"^^xsd:date	
6	http://ontology.ontotext.com/resource/tsnhtmzu9uyo	"Sylvia Mathews Burwell"@en	"1965-06-01"^^xsd:date	

Tip: You can play with more example queries from the `Example_queries.rtf` file provided in the `examples/data/news` directory of the GraphDB distribution.

7.3.3.2 Query data programmatically

SPARQL is not only a standard query language, but also a protocol for communicating with RDF databases. GraphDB stays compliant with the protocol specification, and allows querying data with standard HTTP requests.

Execute the example query with an HTTP GET request:

```
curl -G -H "Accept:application/x-trig" \
  -d query=CONSTRUCT+%7B%3Fs+%3Fp+%3Fo%7D+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10 \
  http://localhost:7200/repositories/yourrepository
```

Execute the example query with a POST operation:

```
curl -X POST --data-binary @file.sparql -H "Accept: application/rdf+xml" \
  -H "Content-type: application/x-www-form-urlencoded" \
  http://localhost:7200/repositories/yourrepository
```

where `file.sparql` contains an encoded query:

```
query=CONSTRUCT+%7B%3Fs+%3Fp+%3Fo%7D+WHERE+%7B%3Fs+%3Fp+%3Fo%7D+LIMIT+10
```

Tip: For more information on how to interact with GraphDB APIs, refer to the RDF4J and SPARQL protocols or the Linked Data Platform specifications.

7.3.4 Visualize GraphDB data with Ogma JS

Ogma is a powerful JavaScript library for graph visualization. In the following examples, data is fetched from a GraphDB repository, converted into an Ogma graph object, and visualized using different graph layouts. All samples reuse functions from a `commons.js` file.

You need a version of Ogma JS to run the samples.

7.3.4.1 People and organizations related to Google in factforge.net

The following example fetches people and organizations related to Google. One of the sample queries in factforge.net is used and rewritten into a CONSTRUCT query. Type is used to differ entities of different types.

```
<html>
<body>
<!-- Include the library -->
<script src="../lib/ogma.min.js"></script>
<script src="../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../lib/lodash.js"></script>
<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole screen. --
↪>
<div id="graph-container" style="position: absolute; left: 0; top: 0; bottom: 0; right: 0;"></div>

<script>
// Which namespace to chose types from
var dboNamespace = "http://dbpedia.org/ontology"

// One of factforge saved queries enriched with types and rdf rank
var peopleAndOrganizationsRelatedToGoogle = `
         # F03: People and organizations related to Google
         # - picks up people related through any type of relationships
         # - picks up parent and child organizations
         # - benefits from inference over transitive dbo:parent
         # - RDFRank makes it easy to see the "top suspects" in a list of 94 entities
         # Change Google with any organization, e.g. type dbr:Hew and Ctrl-Space to auto-complete

         PREFIX dbo: <http://dbpedia.org/ontology/>
         PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
         PREFIX dbr: <http://dbpedia.org/resource/>
         PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
         CONSTRUCT {
             dbr:Google ?p2 ?related_entity .
             dbr:Google sesame:directType ?type .
             ?related_entity ?p1 dbr:Google .
             ?related_entity sesame:directType ?entity_type .
             ?related_entity rank:hasRDFRank ?related_entity_rank .
             dbr:Google dbr:hasChildOrParentOrg ?related_organization .
             ?related_organization sesame:directType ?org_type .
             ?related_organization rank:hasRDFRank ?related_org_rank .
         }
         WHERE {
             BIND( dbr:Google AS ?entity )
             {
               ?related_entity a dbo:Person; ?p1 ?entity .
             FILTER(?p1 NOT IN (dbo:wikiPageWikiLink)) .
               ?related_entity sesame:directType ?entity_type .
               ?related_entity rank:hasRDFRank ?related_entity_rank .
             }
             UNION
             {
               ?related_entity a dbo:Person .
               ?entity ?p2 ?related_entity .
               FILTER(?p2 NOT IN (dbo:wikiPageWikiLink)) .
               ?related_entity sesame:directType ?entity_type .
               ?related_entity rank:hasRDFRank ?related_entity_rank .
             }
             UNION
             {
               ?related_organization a dbo:Organisation ; (dbo:parent | ^dbo:parent) ?entity .
```

(continues on next page)

(continued from previous page)

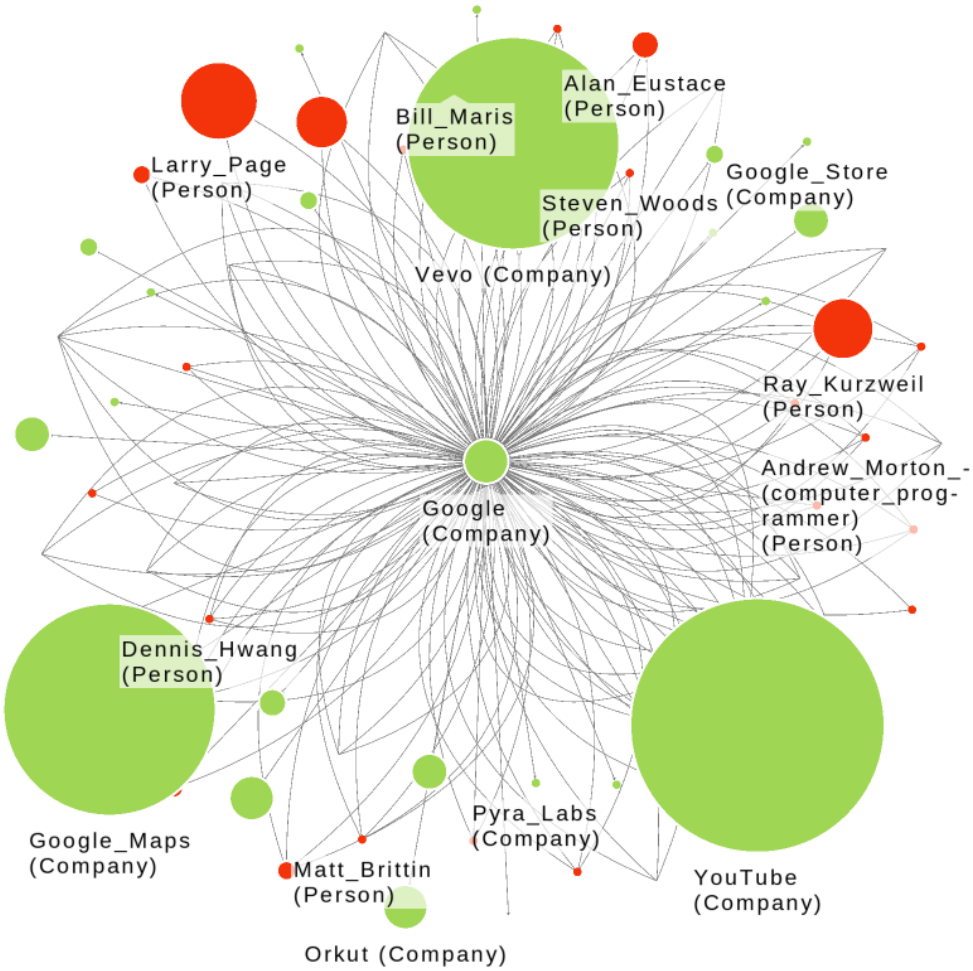
```
                ?related_organization sesame:directType ?org_type .
                ?related_organization rank:hasRDFRank ?related_org_rank .
              } UNION {
                dbr:Google sesame:directType ?type .
              }
          }
          `

var postData = {
        query: peopleAndOrganizationsRelatedToGoogle,
        infer: true,
        sameAs: true,
        limit: 1000,
        offset: 0
}


$.ajax({
        url: graphDBRepoLocation,
        type: 'POST',
        data: postData,
        headers: {
                'Accept': 'application/rdf+json'
        },
        success: function (data) {

                // Converts rdf+json to a simple list of triples
                var triples = convertData(data);

                // Get all nodes uris
                var linkTriples = _.filter(triples, function (triple) {
                        return triple[1] !== rankPredicate && triple[1] !== typePredicate
                });
                var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
                        return t[0]
                }), _.map(linkTriples, function (t) {
                        return t[2]
                })));

                // Get triples for rdf rank
                var ranks = _.filter(triples, function (triple) {
                        return triple[1] === rankPredicate
                });

                // Get triples for types
                var typeTriples = _.filter(triples, function (triple) {
                        return triple[1] === typePredicate && triple[2].indexOf(dboNamespace) === 0
                });

                // Create node objects
                var nodes = _.map(nodesUris, function (nUri) {
                        var rank = _.find(ranks, function (rankTriple) {
                                return rankTriple[0] === nUri && rankTriple[1] === rankPredicate
                        });
                        var type = _.find(typeTriples, function (typeTriple) {
                                return typeTriple[0] === nUri && typeTriple[1] === typePredicate
                        });
                        return {
                                id: nUri,
                                text: getLocalName(nUri) + (type != undefined ? " (" +
↪getLocalName(type[2]) + ")" : ""),
```

(continues on next page)

(continued from previous page)

```
                                size: ((rank != undefined) ? rank[2] * 100 : 5),
                                color: ((type != undefined) ? stringToColour(type[2]) : "#eceeef"),
                        }
                });

                // Create edge objects
                var edges = _.map(linkTriples, function (triple, index) {
                        return {
                                id: index,
                                source: triple[0],
                                target: triple[2],
                                text: getLocalName(triple[1]),
                                shape: 'arrow',
                                size: 0.2
                        }
                });

                // Initialize ogma with the data
                var ogma = new Ogma({
                        container: 'graph-container',
                        settings: {
                                texts: {
                                        nodeFontSize: 20,
                                        edgeFontSize: 15,
                                        nodeSizeThreshold: 0
                                }
                        },
                        graph: {
                                nodes: nodes,
                                edges: edges
                        }
                });

                ogma.locate.center();

                ogma.layouts.start('forceLink', {}, {
                        // sync parameters
                        onEnd: endLayout
                });

                function endLayout() {
                        ogma.locate.center({
                                easing: 'linear',
                                duration: 300
                        });
                }
        }
})
</script>
</body>
</html>
```

Which produces the following graph:



7.3.4.2 Suspicious control chain through off-shore companies in factforge.net

The following example fetches suspicious control chain through off-shore companies, which is another saved query in factforge.net rewritten as a graph query. The entities, their RDF Rank, and their type are fetched. Node size is based on RDF Rank and node color on its type. All examples use a `commons.js` file with some common function, i.e., data model conversion.

```
<html>
<body>
<!-- Include the library -->
<script src="../lib/ogma.min.js"></script>
<script src="../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../lib/lodash.js"></script>

<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole screen. --
↪>
<div id="graph-container" style="position: absolute; left: 0; top: 0; bottom: 0; right: 0;"></div>

<script>

// Which namespace to chose types from
var dboNamespace = "http://dbpedia.org/ontology"
```

(continues on next page)

(continued from previous page)

```
var suspiciousOffshore = `
# F05: Suspicious control chain through off-shore company

PREFIX onto: <http://www.ontotext.com/>
PREFIX fibo-fnd-rel-rel: <http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/>
PREFIX ff-map: <http://factforge.net/ff2016-mapping/>
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
PREFIX dbo: <http://dbpedia.org/ontology/>

CONSTRUCT  {
    ?c1 fibo-fnd-rel-rel:controls ?c2 .
    ?c2 fibo-fnd-rel-rel:controls ?c3 .
    ?c1 ff-map:primaryCountry ?c1_country .
    ?c2 ff-map:primaryCountry ?c2_country .
    ?c3 ff-map:primaryCountry ?c3_country .
    ?c1 sesame:directType ?t1 .
    ?c2 sesame:directType ?t2 .
    ?c3 sesame:directType ?t3 .
    ?c1_country sesame:directType dbo:Country .
    ?c3_country sesame:directType dbo:Country .
    ?c3_country sesame:directType dbo:Country .

} FROM onto:disable-sameAs
WHERE {
    ?c1 fibo-fnd-rel-rel:controls ?c2 .
    ?c2 fibo-fnd-rel-rel:controls ?c3 .
    ?c1 sesame:directType ?t1 .
    ?c2 sesame:directType ?t2 .
    ?c3 sesame:directType ?t3 .
    ?c1 ff-map:primaryCountry ?c1_country .
    ?c2 ff-map:primaryCountry ?c2_country .
    ?c3 ff-map:primaryCountry ?c1_country .
    FILTER (?c1_country != ?c2_country)

    ?c2_country ff-map:hasOffshoreProvisions true .
} `

var postData = {
        query: suspiciousOffshore,
        infer: true,
        sameAs: true,
        limit: 1000,
        offset: 0
}


$.ajax({
        url: graphDBRepoLocation,
        type: 'POST',
        data: postData,
        headers: {
                'Accept': 'application/rdf+json'
        },
        success: function (data) {

                var triples = convertData(data);

                // Get all nodes uris
                var linkTriples = _.filter(triples, function (triple) {
                        return triple[1] !== typePredicate
                });
```

(continues on next page)

(continued from previous page)

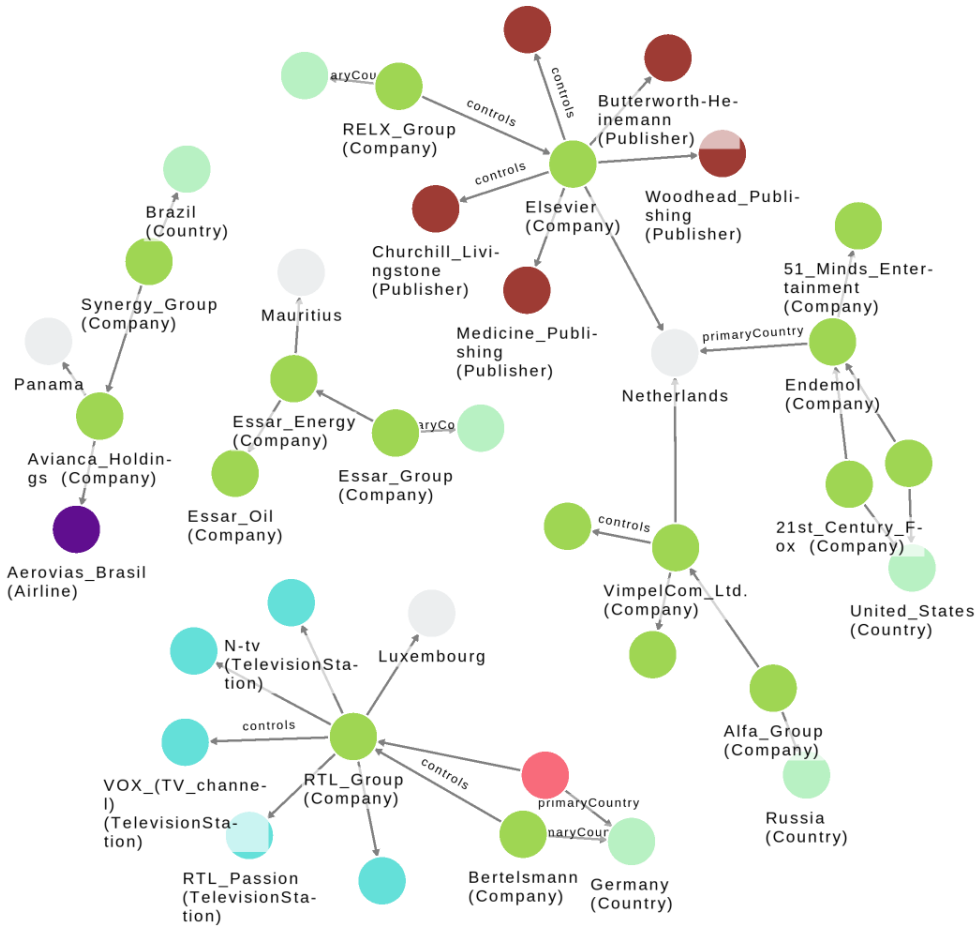
```
                var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
                        return t[0]
                }), _.map(linkTriples, function (t) {
                        return t[2]
                })));

                // Get triples for types
                var typeTriples = _.filter(triples, function (triple) {
                        return triple[1] === typePredicate && triple[2].indexOf(dboNamespace) === 0
                });

                // Create node objects
                var nodes = _.map(nodesUris, function (nUri) {
                        var type = _.find(typeTriples, function (typeTriple) {
                                return typeTriple[0] === nUri && typeTriple[1] === typePredicate
                        });
                        return {
                                id: nUri,
                                text: getLocalName(nUri) + (type != undefined ? " (" +
↪getLocalName(type[2]) + ")" : ""),
                                size: 5,
                                color: ((type != undefined) ? stringToColour(type[2]) : "#eceeef"),
                        }
                });

                // Create edge objects
                var edges = _.map(linkTriples, function (triple, index) {
                        return {
                                id: index,
                                source: triple[0],
                                target: triple[2],
                                text: getLocalName(triple[1]),
                                shape: 'arrow',
                                size: 0.5
                        }
                });

                // Initialize ogma with the data
                var ogma = new Ogma({
                        container: 'graph-container',
                        settings: {
                                texts: {
                                        nodeFontSize: 20,
                                        edgeFontSize: 15,
                                        nodeSizeThreshold: 0,
                                        edgeSizeThreshold: 0
                                }
                        },
                        graph: {
                                nodes: nodes,
                                edges: edges
                        }
                });

                ogma.locate.center();

                ogma.layouts.start('forceLink', {}, {
                        onEnd: endLayout
                });

                function endLayout() {
```

(continues on next page)

(continued from previous page)

```
                        ogma.locate.center({
                                easing: 'linear',
                                duration: 300
                        });
                }
        }
})
</script>
</body>
</html>
```

Which produces the following graph:



7.3.4.3 Shortest flight path

1. Import the `airports.ttl` dataset, which contains airports and flights.
2. Display the airports on a map using the latitude and longitude properties.
3. Find the shortest path between airports in terms of number of flights.

```
<html>
<body>
<!-- Include the library -->
<script src="../lib/ogma.min.js"></script>
<script src="../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../lib/lodash.js"></script>
```

(continues on next page)

(continued from previous page)

```
  <style>
    #graph-container { top: 0; bottom: 0; left: 0; right: 0; position: absolute; margin: 0; overflow:
↪hidden; }
    .info {
      position: absolute;
      color: #fff;
      background: #141229;
      font-size: 12px;
      font-family: monospace;
      padding: 5px;
    }
    .info.n { top: 0; left: 0; }
  </style>

<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole screen. --
↪>
<div id="graph-container"></div>
<div id="n" class="info n">loading a large graph, it can take a few seconds...</div>

<script>

// The query to visualize
var airportsQuery = `
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
construct {
    ?source <http://openflights.org/resource/route/hasFlightTo> ?dest.
    ?dest rdf:type ?dtype .
    ?dest rdfs:label ?destLabel .
    ?source rdf:type ?stype .
    ?source rdfs:label ?sLabel .
    ?source <http://openflights.org/resource/airport/latitide> ?sourceLat .
    ?dest <http://openflights.org/resource/airport/latitide> ?destLat .
    ?source <http://openflights.org/resource/airport/longtitude> ?sourceLong .
    ?dest <http://openflights.org/resource/airport/longtitude> ?destLong .
} where {
    ?flight <http://openflights.org/resource/route/destinationId> ?dest .
    ?flight <http://openflights.org/resource/route/sourceId> ?source .
    ?flight rdf:type ?ftype .
    ?dest rdf:type ?dtype .
    ?dest rdfs:label ?destLabel .
    ?source rdf:type ?stype .
    ?source rdfs:label ?sLabel .
    ?source <http://openflights.org/resource/airport/latitide> ?sourceLat .
    ?dest <http://openflights.org/resource/airport/latitide> ?destLat .
    ?source <http://openflights.org/resource/airport/longtitude> ?sourceLong .
    ?dest <http://openflights.org/resource/airport/longtitude> ?destLong .
}
`;

var typePredicate = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type";
var labelPredicate = "http://www.w3.org/2000/01/rdf-schema#label";
var latitudePredicate = "http://openflights.org/resource/airport/latitide";
var longtitudePredicate = "http://openflights.org/resource/airport/longtitude";


var postData = {
        query: airportsQuery,
        infer: true,
        sameAs: true,
```

(continues on next page)

(continued from previous page)

```
        // limit: 10000
}

var startNode = 'http://openflights.org/resource/airport/id/1194';
var endNode = 'http://openflights.org/resource/airport/id/4061';


$.ajax({
        url: 'http://localhost:8082/repositories/airroutes',
        type: 'POST',
        data: postData,
        headers: {
                'Accept': 'application/rdf+json'
        },
        success: function (data) {

                var triples = convertData(data);

                // Get all nodes uris
                var linkTriples = _.filter(triples, function (triple) {
                        return triple[1] !== typePredicate && triple[1] !== labelPredicate && triple[1]
↪!= latitudePredicate && triple[1] != longtitudePredicate
                });
                var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
                        return t[0]
                }), _.map(linkTriples, function (t) {
                        return t[2]
                })));

                // Get triples for types
                var typeTriples = _.filter(triples, function (triple) {
                        return triple[1] === typePredicate
                });
                var labelTriples = _.filter(triples, function (triple) {
                        return triple[1] === labelPredicate
                });
                var latitudeTriples = _.filter(triples, function (triple) {
                        return triple[1] === latitudePredicate
                });
                var longtitudeTriples = _.filter(triples, function (triple) {
                        return triple[1] === longtitudePredicate
                });

                // Create node objects
                var nodes = _.map(nodesUris, function (nUri) {
                        var type = _.find(typeTriples, function (typeTriple) {
                                return typeTriple[0] === nUri && typeTriple[1] === typePredicate
                        });
                        var label = _.find(labelTriples, function (labelTriple) {
                                return labelTriple[0] === nUri && labelTriple[1] === labelPredicate
                        });
                        var latitude = _.find(latitudeTriples, function (latTriple) {
                                return latTriple[0] === nUri && latTriple[1] === latitudePredicate
                        });
                        var longtitude = _.find(longtitudeTriples, function (longTriple) {
                                return longTriple[0] === nUri && longTriple[1] === longtitudePredicate
                        });
                        return {
                                id: nUri,
                                text: (label != undefined) ? (label[2] + "(" + getLocalName(nUri) + ")
↪") : getLocalName(nUri),
```

(continues on next page)

(continued from previous page)

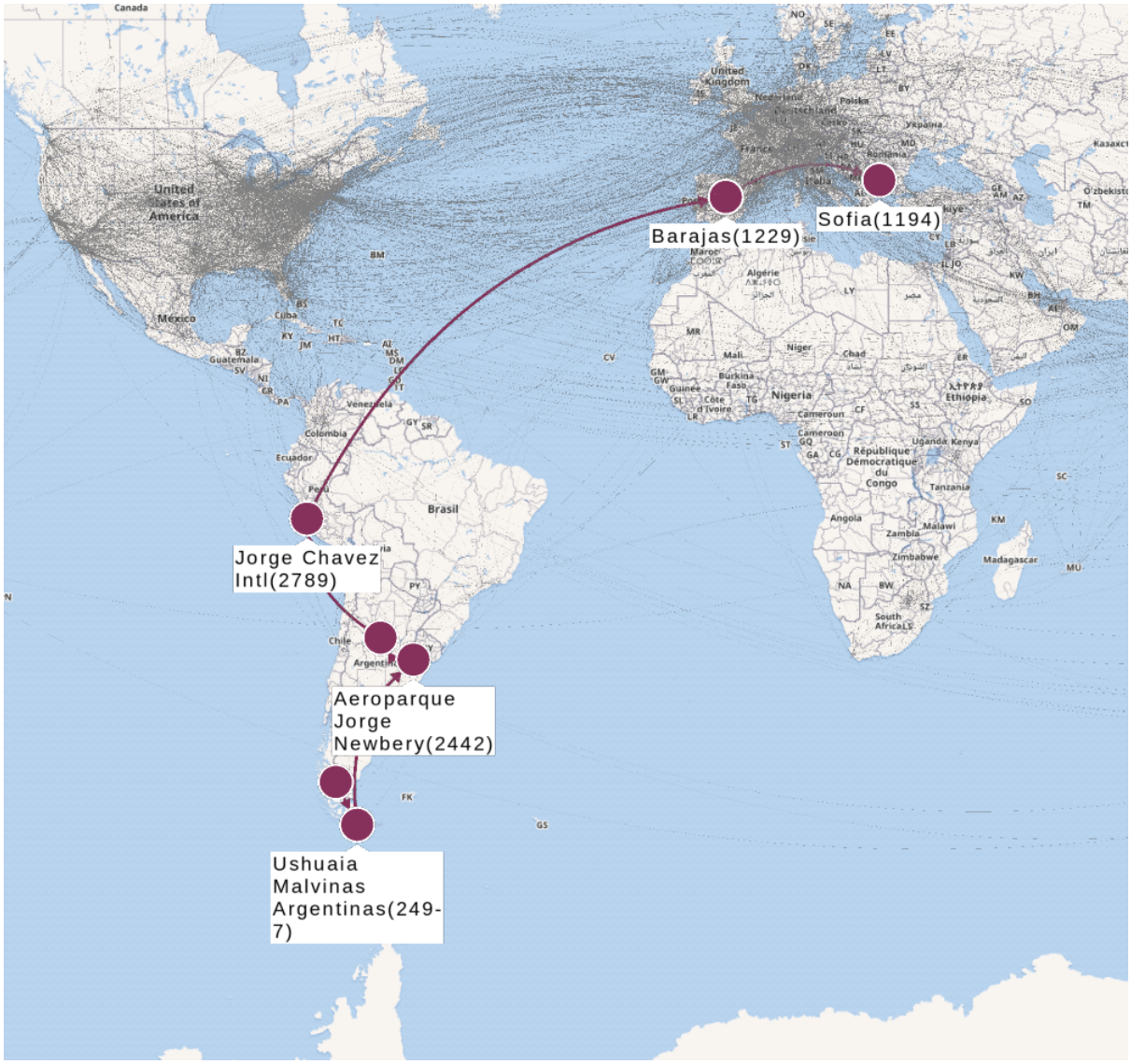
```
                                size: 0.5,
                                color: ((type != undefined) ? stringToColour(type[2]) : "#eceeef"),
                                latitude: (latitude != undefined) ? parseFloat(latitude[2]) : 0,
                                longitude: (longtitude != undefined) ? parseFloat(longtitude[2]) : 0,
                        }
                });

                // Create edge objects
                var edges = _.map(linkTriples, function (triple, index) {
                        return {
                                id: index,
                                source: triple[0],
                                target: triple[2],
                                text: getLocalName(triple[1]),
                                shape: 'arrow',
                                size: 0.5
                        }
                });

                var url = Ogma.utils.pixelRatio() === 2 ? // retina displays
                        'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}@2x.png' :
                        'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png';

                // Initialize ogma with the data
                var ogma = new Ogma({
                        container: 'graph-container',
                        settings: {
                                geo: {
                                        tileUrlTemplate: url, // indicates from which server the tiles
↪must be retrieved
                                        sizeZoomReferential: 5, // Paris will be displayed with a radius
↪of 8 pixels on the screen if the geographical zoom is 5
                                        attribution: '<div class="attribution">Map data © <a target="_
↪blank" href="http://osm.org/copyright">OpenStreetMap contributors</a></div>'
                                },
                                texts: {
                                        nodeFontSize: 20,
                                        edgeFontSize: 15,
                                        nodeBackgroundColor: '#fff',
                                }
                        },
                        graph: {
                                nodes: nodes,
                                edges: edges
                        }
                });

                ogma.geo.enable();

                var pathNodes = ogma.pathfinding.dijkstra(startNode, endNode);
                if (pathNodes) {
                        var ids = pathNodes.map(function (node) {
                                return node.id
                        });

                        // Color the path
                        for (var i = 0; i < pathNodes.length; i++) {
                                pathNodes[i].color = '#86315b';
                                pathNodes[i].size = 2;

                                ogma.topology.getAdjacentEdges(pathNodes[i]).forEach(function (edge) {
```

(continues on next page)

(continued from previous page)

```
                                        if (ids.indexOf(edge.source) != -1 && ids.indexOf(edge.target) !
↪= -1 && ids.indexOf(edge.source) < ids.indexOf(edge.target)) {
                                                edge.color = '#86315b';
                                                edge.size = 0.4
                                        }
                                }});
                        }
                }

                document.getElementById('n').textContent = 'nodes: ' + ogma.graph.nodes.length + ';
↪edges: ' + ogma.graph.edges.length;
        }
})

</script>
</body>
</html>
```

Which produces the following graph:



7.3.4.4 Common function to visualize GraphDB data

The `commons.js` file used by all demos:

```
var stringToColour = function(str) {
    var hash = 0;
    for (var i = 0; i < str.length; i++) {
        hash = str.charCodeAt(i) + ((hash << 5) - hash);
    }
    var colour = '#';
    for (var i = 0; i < 3; i++) {
        var value = (hash >> (i * 8)) & 0xFF;
        colour += ('00' + value.toString(16)).substr(-2);
    }
    return colour;
}

var getLocalName = function(str) {
    return str.substr(Math.max(str.lastIndexOf('/'), str.lastIndexOf('#')) + 1);
}

var getPrefix = function(str) {
    return str.substr(0, Math.max(str.lastIndexOf('/'), str.lastIndexOf('#')));
}

var convertData = function(data) {
    var mapped = _.map(data, function(value, subject) {
        return _.map(value, function(value1, predicate) {
            return _.map(value1, function(object) {
                return [
                    subject,
                    predicate,
                    object.value
                ]
            })
        })
    });

    // Convert graph json to array of triples
    var triples = _.reduce(mapped, function(memo, el) {
        return memo.concat(el)
    }, []);
    triples = _.reduce(triples, function(memo, el) {
        return memo.concat(el)
    }, []);
    return triples;
}


// The RDFRank, nodes size is calculated according to RDFRank
var rankPredicate = "http://www.ontotext.com/owlim/RDFRank#hasRDFRank";

// Get type for a node to color nodes of the same type with the same color
var typePredicate = "http://www.openrdf.org/schema/sesame#directType";

// The location of a graphdb repo endpoint
var graphDBRepoLocation = 'http://factforge.net/repositories/ff-news';
```

Learn more about linkurious and ogma.

7.4 GraphDB and W3C standards

7.4.1 Introduction to the Semantic Web

The Semantic Web represents a broad range of ideas and technologies that attempt to bring meaning to the vast amount of information available via the Web. The intention is to provide information in a structured form so that it can be processed automatically by machines. The combination of structured data and inferencing can yield much information not explicitly stated.

The aim of the Semantic Web is to solve the most problematic issues that come with the growth of the non-semantic (HTML-based or similar) Web that results in a high level of human effort for finding, retrieving and exploiting information. For example, contemporary search engines are extremely fast, but tend to be very poor at producing relevant results. Of the thousands of matches typically returned, only a few point to truly relevant content and some of this content may be buried deep within the identified pages. Such issues dramatically reduce the value of the information discovered as well as the ability to automate the consumption of such data. Other problems related to classification and generalization of identifiers further confuse the landscape.

The Semantic Web solves such issues by adopting unique identifiers for concepts and the relationships between them. These identifiers, called *Universal Resource Identifiers (URIs)* (a "resource" is any 'thing' or 'concept') are similar to Web page URLs, but do not necessarily identify documents from the Web. Their sole purpose is to uniquely identify objects or concepts and the relationships between them.

The use of URIs removes much of the ambiguity from information, but the Semantic Web goes further by allowing concepts to be associated with hierarchies of classifications, thus making it possible to infer new information based on an individual's classification and relationship to other concepts. This is achieved by making use of *ontologies* — hierarchical structures of concepts — to classify individual concepts.

7.4.1.1 Resource Description Framework (RDF)

The World-Wide Web has grown rapidly and contains huge amounts of information that cannot be interpreted by machines. Machines cannot understand meaning, therefore they cannot understand Web content. For this reason, most attempts to retrieve some useful pieces of information from the Web require a high degree of user involvement — manually retrieving information from multiple sources (different Web pages), 'digging' through multiple search engine results (where useful pieces of data are often buried many pages deep), comparing differently structured result sets (most of them incomplete), and so on.

For the machine interpretation of semantic content to become possible, there are two prerequisites:

1. Every concept should be uniquely identified. (For example, if a particular person owns a web site, authors articles on other sites, gives an interview on another site and has profiles in a couple of social media sites such as Facebook and LinkedIn, then the occurrences of his name/identifier in all these places should be related to exact same identifier.)
2. There must be a unified system for conveying and interpreting meaning that all automated search agents and data storage applications should use.

One approach for attaching semantic information to Web content is to embed the necessary machine-processable information through the use of special meta-descriptors (meta-tagging) in addition to the existing meta-tags that mainly concern the layout.

Within these meta tags, the *resources* (the pieces of useful information) can be uniquely identified in the same manner in which Web pages are uniquely identified, i.e., by extending the existing URL system into something more universal — a URI (Uniform Resource Identifier). In addition, conventions can be devised, so that resources can be described in terms of properties and values (resources can have properties and properties have values). The concrete implementations of these conventions (or vocabularies) can be embedded into Web pages (through meta-descriptors again) thus effectively 'telling' the processing machines things like:

[resource] **John Doe** has a [property] **web site** which is [value] **www.johndoesite.com**

The *Resource Description Framework (RDF)* developed by the World Wide Web Consortium (W3C) makes possible the automated semantic processing of information, by structuring information using individual *statements* that

consist of: Subject, Predicate, Object. Although frequently referred to as a 'language', RDF is mainly a data model. It is based on the idea that the things being described have properties, which have values, and that resources can be described by making statements. RDF prescribes how to make statements about resources, in particular, Web resources, in the form of subject-predicate-object expressions. The 'John Doe' example above is precisely this kind of statement. The statements are also referred to as *triples*, because they always have the subject-predicate-object structure.

The basic RDF components include statements, Uniform Resource Identifiers, properties, blank nodes, and literals. RDF-star (formerly RDF*) extends RDF with support for embedded triples. They are discussed in the topics that follow.

Uniform Resource Identifiers (URIs)

A unique Uniform Resource Identifier (URI) is assigned to any resource or thing that needs to be described. Resources can be authors, books, publishers, places, people, hotels, goods, articles, search queries, and so on. In the Semantic Web, every resource has a URI. A URI can be a URL or some other kind of unique identifier. Unlike URLs, URIs do not necessarily enable access to the resource they describe, i.e, in most cases they do not represent actual web pages. For example, the string `http://www.johndoesite.com/aboutme.htm`, if used as a URL (Web link) is expected to take us to a Web page of the site providing information about the site owner, the person John Doe. The same string can however be used simply to identify that person on the Web (URI) irrespective of whether such a page exists or not.

Thus URI schemes can be used not only for Web locations, but also for such diverse objects as telephone numbers, ISBN numbers, and geographic locations. In general, we assume that a URI is the identifier of a resource and can be used as either the subject or the object of a statement. Once the subject is assigned a URI, it can be treated as a resource and further statements can be made about it.

This idea of using URIs to identify 'things' and the relations between them is important. This approach goes some way towards a global, unique naming scheme. The use of such a scheme greatly reduces the homonym problem that has plagued distributed data representation in the past.

Statements: Subject-Predicate-Object triples

To make the information in the following sentence

"The web site www.johndoesite.com is created by John Doe."

machine-accessible, it should be expressed in the form of an RDF statement, i.e., a subject-predicate-object triple:

"[subject] the web site www.johndoesite.com [predicate] has a creator [object] called John Doe."

This statement emphasizes the fact that in order to describe something, there has to be a way to name or identify a number of things:

- the thing the statement describes (Web site "www.johndoesite.com");
- a specific property ("creator") of the thing the statement describes;
- the thing the statement says is the value of this property (who the owner is).

The respective RDF terms for the various parts of the statement are:

- the subject is the URL "www.johndoesite.com";
- the predicate is the expression "has creator";
- the object is the name of the creator, which has the value "John Doe".

Next, each member of the subject-predicate-object triple should be identified using its URI, e.g.:

- the subject is `http://www.johndoesite.com`;
- the predicate is `http://purl.org/dc/elements/1.1/creator` (this is according to a particular RDF Schema, namely, the Dublin Core Metadata Element Set);

- the object is `http://www.johndoesite.com/aboutme` (which may not be an actual web page).

Note that in this version of the statement, instead of identifying the creator of the web site by the character string "John Doe", we used a URI, namely `http://www.johndoesite.com/aboutme`. An advantage of using a URI is that the identification of the statement's subject can be more precise, i.e., the creator of the page is neither the character string "John Doe", nor any of the thousands of other people with that name, but the particular John Doe associated with this URI (whoever created the URI defines the association). Moreover, since there is a URI to refer to John Doe, he is now a full-fledged resource and additional information can be recorded about him simply by adding additional RDF statements with John's URI as the subject.

What we basically have now is the logical formula $P(x, y)$, where the binary predicate P relates the object x to the object y — we may also think of this formula as written in the form x, P, y. In fact, RDF offers only binary predicates (properties). If more complex relationships are to be defined, this is done through sets of multiple RDF triples. Therefore, we can describe the statement as:

```
<http://www.johndoesite.com> <http://purl.org/dc/elements/1.1/creator> <http://www.johndoesite.com/
↪aboutme>
```

There are several conventions for writing abbreviated RDF statements, as used in the RDF specifications themselves. This shorthand employs an XML *qualified name* (or *QName*) without angle brackets as an abbreviation for a full URI reference. A QName contains a prefix that has been assigned to a namespace URI, followed by a colon, and then a local name. The full URI reference is formed from the QName by appending the local name to the namespace URI assigned to the prefix. So, for example, if the QName prefix `foo` is assigned to the namespace URI `http://example.com/somewhere/`, then the QName "foo:bar" is a shorthand for the URI `http://example.com/somewhere/bar`.

In our example, we can define the namespace `jds` for `http://www.johndoesite.com` and use the Dublin Core Metadata namespace `dc` for `http://purl.org/dc/elements/1.1/`.

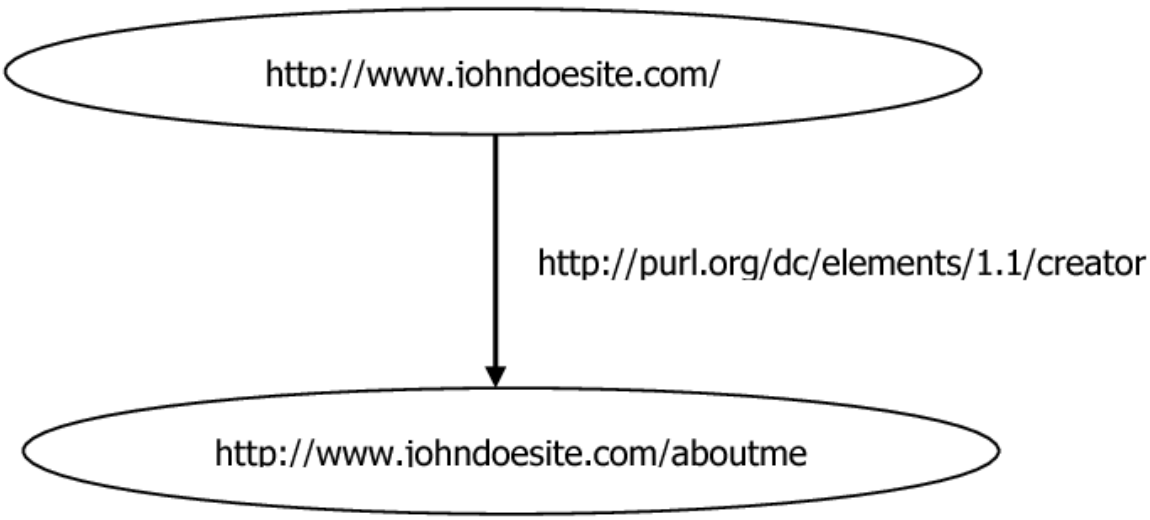
So, the shorthand form for the example statement is simply:

```
jds: dc:creator jds:aboutme
```

Objects of RDF statements can (and very often do) form the subjects of other statements leading to a graph-like representation of knowledge. Using this notation, a statement is represented by:

- a node for the subject;
- a node for the object;
- an arc for the predicate, directed from the subject node to the object node.

So the RDF statement above could be represented by the following graph:



This kind of graph is known in the artificial intelligence community as a 'semantic net'.

In order to represent RDF statements in a machine-processable way, RDF uses mark-up languages, namely (and almost exclusively) the Extensible Mark-up Language (XML). Because an abstract data model needs a concrete syntax in order to be represented and transmitted, RDF has been given a syntax in XML. As a result, it inherits the benefits associated with XML. However, it is important to understand that other syntactic representations of RDF, not based on XML, are also possible. XML-based syntax is not a necessary component of the RDF model. XML was designed to allow anyone to design their own document format and then write a document in that format. RDF defines a specific XML mark-up language, referred to as RDF/XML, for use in representing RDF information and for exchanging it between machines. Written in RDF/XML, our example will look as follows:

```
<?xml version="1.0" encoding="UTF-16"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:jds="http://www.johndoesite.com/"
        xmlns:dc="http://purl.org/dc/elements/1.1/">
        <rdf:Description rdf:about="http://www.johndoesite.com/">
            <dc:creator rdf:resource="jds:aboutme"/>
        </rdf:Description>
</rdf:RDF>
```

Note: RDF/XML uses the namespace mechanism of XML, but in an expanded way. In XML, namespaces are only used for disambiguation purposes. In RDF/XML, external namespaces are expected to be RDF documents defining resources, which are then used in the importing RDF document. This mechanism allows the reuse of resources by other people who may decide to insert additional features into these resources. The result is the emergence of large, distributed collections of knowledge.

Also observe that the `rdf:about` attribute of the element `rdf:Description` is equivalent in meaning to that of an ID attribute, but it is often used to suggest that the object about which a statement is made has already been 'defined' elsewhere. Strictly speaking, a set of RDF statements together simply forms a large graph, relating things to other things through properties, and there is no such concept as 'defining' an object in one place and referring to it elsewhere. Nevertheless, in the serialized XML syntax, it is sometimes useful (if only for human readability) to suggest that one location in the XML serialization is the 'defining' location, while other locations state 'additional' properties about an object that has been 'defined' elsewhere.

Properties

Properties are a special kind of resource: they describe relationships between resources, e.g., `written by`, `age`, `title`, and so on. Properties in RDF are also identified by URIs (in most cases, these are actual URLs). Therefore, properties themselves can be used as the subject in other statements, which allows for an expressive ways to describe properties, e.g., by defining property hierarchies.

Named graphs

A named graph (NG) is a set of triples named by a URI. This URI can then be used outside or within the graph to refer to it. The ability to name a graph allows separate graphs to be identified out of a large collection of statements and further allows statements to be made about graphs.

Named graphs represent an extension of the RDF data model, where quadruples `<s,p,o,ng>` are used to define statements in an RDF multi-graph. This mechanism allows, e.g., the handling of provenance when multiple RDF graphs are integrated into a single repository.

From the perspective of GraphDB, named graphs are important, because comprehensive support for SPARQL requires NG support.

7.4.1.2 RDF Schema (RDFS)

While being a universal model that lets users describe resources using their own vocabularies, RDF does not make assumptions about any particular application domain, nor does it define the semantics of any domain. It is up to the user to do so using an *RDF Schema (RDFS)* vocabulary.

RDF Schema is a vocabulary description language for describing properties and classes of RDF resources, with a semantics for generalization hierarchies of such properties and classes. Be aware of the fact that the RDF Schema is conceptually different from the XML Schema, even though the common term *schema* suggests similarity. The XML Schema constrains the structure of XML documents, whereas the RDF Schema defines the vocabulary used in RDF data models. Thus, RDFS makes semantic information machine-accessible, in accordance with the Semantic Web vision. RDF Schema is a primitive ontology language. It offers certain modelling primitives with fixed meaning.

RDF Schema does not provide a vocabulary of application-specific classes. Instead, it provides the facilities needed to describe such classes and properties, and to indicate which classes and properties are expected to be used together (for example, to say that the property `JobTitle` will be used in describing a class "Person"). In other words, RDF Schema provides a type system for RDF.

The RDF Schema type system is similar in some respects to the type systems of object-oriented programming languages such as Java. For example, RDFS allows resources to be defined as instances of one or more classes. In addition, it allows classes to be organized in a hierarchical fashion. For example, a class `Dog` might be defined as a subclass of `Mammal`, which itself is a subclass of `Animal`, meaning that any resource that is in class `Dog` is also implicitly in class `Animal` as well.

RDF classes and properties, however, are in some respects very different from programming language types. RDF class and property descriptions do not create a straight-jacket into which information must be forced, but instead provide additional information about the RDF resources they describe.

The RDFS facilities are themselves provided in the form of an RDF vocabulary, i.e., as a specialized set of predefined RDF resources with their own special meanings. The resources in the RDFS vocabulary have URIs with the prefix `http://www.w3.org/2000/01/rdf-schema#` (conventionally associated with the namespace prefix `rdfs`). Vocabulary descriptions (schemas) written in the RDFS language are legal RDF graphs. Hence, systems processing RDF information that do *not* understand the additional RDFS vocabulary can still interpret a schema as a legal RDF graph consisting of various resources and properties. However, such a system will be oblivious to the additional built-in meaning of the RDFS terms. To understand these additional meanings, the software that processes RDF information has to be extended to include these language features and to interpret their meanings in the defined way.

Describing classes

A class can be thought of as a set of elements. Individual objects that belong to a class are referred to as instances of that class. A class in RDFS corresponds to the generic concept of a type or category similar to the notion of a class in object-oriented programming languages such as Java. RDF classes can be used to represent any category of objects such as web pages, people, document types, databases or abstract concepts. Classes are described using the RDF Schema resources `rdfs:Class` and `rdfs:Resource`, and the properties `rdf:type` and `rdfs:subClassOf`. The relationship between instances and classes in RDF is defined using `rdf:type`.

An important use of classes is to impose restrictions on what can be stated in an RDF document using the schema. In programming languages, typing is used to prevent incorrect use of objects (resources) and the same is true in RDF imposing a restriction on the objects to which the property can be applied. In logical terms, this is a restriction on the domain of the property.

Describing properties

In addition to describing the specific classes of things they want to describe, user communities also need to be able to describe specific properties that characterize these classes of things (such as `numberOfBedrooms` to describe an apartment). In RDFS, properties are described using the RDF class `rdf:Property`, and the RDFS properties `rdfs:domain`, `rdfs:range` and `rdfs:subPropertyOf`.

All properties in RDF are described as instances of class `rdf:Property`. So, a new property, such as `exterms:weightInKg`, is defined with the following RDF statement:

```
exterms:weightInKg   rdf:type   rdf:Property .
```

RDFS also provides vocabulary for describing how properties and classes are intended to be used together. The most important information of this kind is supplied by using the RDFS properties `rdfs:range` and `rdfs:domain` to further describe application-specific properties.

The `rdfs:range` property is used to indicate that the values of a particular property are members of a designated class. For example, to indicate that the property `ex:author` has values that are instances of class `ex:Person`, the following RDF statements are used:

```
ex:Person   rdf:type     rdfs:Class .
ex:author   rdf:type     rdf:Property .
ex:author   rdfs:range   ex:Person .
```

These statements indicate that `ex:Person` is a class, `ex:author` is a property, and that RDF statements using the `ex:author` property have instances of `ex:Person` as objects.

The `rdfs:domain` property is used to indicate that a particular property is used to describe a specific class of objects. For example, to indicate that the property `ex:author` applies to instances of class `ex:Book`, the following RDF statements are used:

```
ex:Book     rdf:type      rdfs:Class .
ex:author   rdf:type      rdf:Property .
ex:author   rdfs:domain   ex:Book .
```

These statements indicate that `ex:Book` is a class, `ex:author` is a property, and that RDF statements using the `ex:author` property have instances of `ex:Book` as subjects.

Sharing vocabularies

RDFS provides the means to create custom vocabularies. However, it is generally easier and better practice to use an existing vocabulary created by someone else who has already been describing a similar conceptual domain. Such publicly available vocabularies, called 'shared vocabularies', are not only cost-efficient to use, but they also promote the shared understanding of the described domains.

Considering the earlier example, in the statement:

```
jds:   dc:creator   jds:aboutme .
```

the predicate `dc:creator`, when fully expanded into a URI, is an unambiguous reference to the `creator` attribute in the Dublin Core metadata attribute set, a widely used set of attributes (properties) for describing information of this kind. So this triple is effectively saying that the relationship between the website (identified by `http://www.johndoesite.com/`) and the creator of the site (a distinct person, identified by `http://www.johndoesite.com/aboutme`) is exactly the property identified by `http://purl.org/dc/elements/1.1/creator`. This way, anyone familiar with the Dublin Core vocabulary or those who find out what `dc:creator` means (say, by looking up its definition on the Web) will know what is meant by this relationship. In addition, this shared understanding based upon using unique URIs for identifying concepts is exactly the requirement for creating computer systems that can automatically process structured information.

However, the use of URIs does not solve all identification problems, because different URIs can be created for referring to the same thing. For this reason, it is a good idea to have a preference towards using terms from existing

vocabularies (such as the Dublin Core) where possible, rather than making up new terms that might overlap with those of some other vocabulary. Appropriate vocabularies for use in specific application areas are being developed all the time, but even so, the sharing of these vocabularies in a common 'Web space' provides the opportunity to identify and deal with any equivalent terminology.

Dublin Core Metadata Initiative

An example of a shared vocabulary that is readily available for reuse is The Dublin Core, which is a set of elements (properties) for describing documents (and hence, for recording metadata). The element set was originally developed at the March 1995 Metadata Workshop in Dublin, Ohio, USA. Dublin Core has subsequently been modified on the basis of later Dublin Core Metadata workshops and is currently maintained by the Dublin Core Metadata Initiative.

The goal of Dublin Core is to provide a minimal set of descriptive elements that facilitate the description and the automated indexing of document-like networked objects, in a manner similar to a library card catalogue. The Dublin Core metadata set is suitable for use by resource discovery tools on the Internet, such as Web crawlers employed by search engines. In addition, Dublin Core is meant to be sufficiently simple to be understood and used by the wide range of authors and casual publishers of information to the Internet.

Dublin Core elements have become widely used in documenting Internet resources (the Dublin Core `creator` element was used in the earlier examples). The current elements of Dublin Core contain definitions for properties such as `title` (a name given to a resource), `creator` (an entity primarily responsible for creating the content of the resource), `date` (a date associated with an event in the life-cycle of the resource) and `type` (the nature or genre of the content of the resource).

Information using Dublin Core elements may be represented in any suitable language (e.g., in HTML meta elements). However, RDF is an ideal representation for Dublin Core information. The following example uses Dublin Core by itself to describe an audio recording of a guide to growing rose bushes:

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:dc="http://purl.org/dc/elements/1.1/">

    <rdf:Description rdf:about="http://media.example.com/audio/guide.ra">
        <dc:creator>Mr. Dan D. Lion</dc:creator>
        <dc:title>A Guide to Growing Roses</dc:title>
        <dc:description>Describes planting and nurturing rose bushes.
        </dc:description>
        <dc:date>2001-01-20</dc:date>
    </rdf:Description>
</rdf:RDF>
```

The same RDF statements in Notation-3:

```
@prefix dc: <[http://purl.org/dc/elements/1.1/]> .
@prefix rdf: <[http://www.w3.org/1999/02/22-rdf-syntax-ns#]> .

<http://media.example.com/audio/guide.ra> dc:creator "Mr. Dan D. Lion" ;
    dc:title "A Guide to Growing Roses" ;
    dc:description "Describes planting and nurturing rose bushes." ;
    dc:date "2001-01-20" .
```

7.4.1.3 Ontologies and knowledge bases

In general, an ontology formally describes a (usually finite) domain of related concepts (classes of objects) and their relationships. For example, in a company setting, staff members, managers, company products, offices, and departments might be some important concepts. The relationships typically include hierarchies of classes. A hierarchy specifies a class `C` to be a subclass of another class `C'` if every object in `C` is also included in `C'`. For example, all managers are staff members.

Apart from subclass relationships, ontologies may include information such as:

- properties (`X` is subordinated `Y`);
- value restrictions (only managers may head departments);
- disjointness statements (managers and general employees are disjoint);
- specifications of logical relationships between objects (every department must have at least three staff members).

Ontologies are important because *semantic repositories* use ontologies as semantic schemata. This makes automated reasoning about the data possible (and easy to implement) since the most essential relationships between the concepts are built into the ontology.

Formal *knowledge representation* (KR) is about building models. The typical modeling paradigm is mathematical logic, but there are also other approaches, rooted in the information and library science. KR is a very broad term; here we only refer to the mainstream meaning of the world (of a particular state of affairs, situation, domain or problem), which allow for automated reasoning and interpretation. Such models consist of ontologies defined in a formal language. Ontologies can be used to provide formal semantics (i.e., machine-interpretable meaning) to any sort of information: databases, catalogues, documents, Web pages, etc. Ontologies can be used as semantic frameworks: the association of information with ontologies makes such information much more amenable to machine processing and interpretation. This is because ontologies are described using logical formalisms, such as *OWL*, which allow automatic inferencing over these ontologies and datasets that use them, i.e., as a vocabulary.

An important role of ontologies is to serve as schemata or 'intelligent' views over information resources. This is also the role of ontologies in the Semantic Web. Thus, they can be used for indexing, querying, and reference purposes over non-ontological datasets and systems such as databases, document and catalogue management systems. Because ontological languages have formal semantics, ontologies allow a wider interpretation of data, i.e., inference of facts, which are not explicitly stated. In this way, they can improve the interoperability and the efficiency of using arbitrary datasets.

An ontology `O` can be defined as comprising the 4-tuple.

```
O = <C,R,I,A>
```

where

- `C` is a set of classes representing concepts from the domain we wish to describe (e.g., invoices, payments, products, prices, etc);
- `R` is a set of relations (also referred to as properties or predicates) holding between (instances of) these classes (e.g., Product *hasPrice* Price);
- `I` is a set of instances, where each instance can be a member of one or more classes and can be linked to other instances or to literal values (strings, numbers and other data-types) by relations (e.g., `product23 compatibleWith product348` or `product23 hasPrice €170`);
- `A` is a set of axioms (e.g., if a product has a price greater than €200, then shipping is free).

Classification of ontologies

Ontologies can be classified as light-weight or heavy-weight according to the complexity of the KR language and the extent to which it is used. Light-weight ontologies allow for more efficient and scalable reasoning, but do not possess the highly predictive (or restrictive) power of more powerful KR languages. Ontologies can be further differentiated according to the sort of conceptualization that they formalize: upper-level ontologies model general knowledge, while domain and application ontologies represent knowledge about a specific domain (e.g., medicine or sport) or a type of application, e.g., knowledge management systems.

Finally, ontologies can be distinguished according to the sort of semantics being modeled and their intended usage. The major categories from this perspective are:

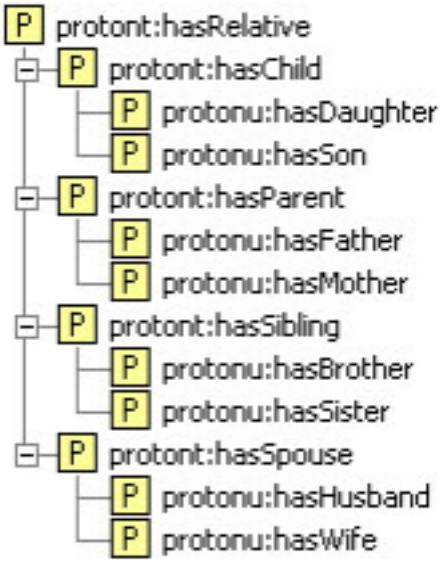
- Schema-ontologies: ontologies that are close in purpose and nature to database and object-oriented schemata. They define classes of objects, their properties and relationships to objects of other classes. A typical use of such an ontology involves using it as a vocabulary for defining large sets of instances. In basic terms, a class in a schema ontology corresponds to a table in a relational database; a relation — to a column; an instance — to a row in the table for the corresponding class;
- Topic-ontologies: taxonomies that define hierarchies of topics, subjects, categories, or designators. These have a wide range of applications related to classification of different things (entities, information resources, files, Web pages, etc). The most popular examples are library classification systems and taxonomies, which are widely used in the knowledge management field. Yahoo and DMOZ are popular large-scale incarnations of this approach. A number of the most popular taxonomies are listed as encoding schemata in Dublin Core;
- Lexical ontologies: lexicons with formal semantics that define lexical concepts. We use 'lexical concept' here as some kind of a formal representation of the meaning of a word or a phrase. In WordNet, for example, lexical concepts are modeled as synsets (synonym sets), while word-sense is the relation between a word and a synset, word-senses and terms. These can be considered as semantic thesauri or dictionaries. The concepts defined in such ontologies are not instantiated, rather they are directly used for reference, e.g., for annotation of the corresponding terms in text. WordNet is the most popular general purpose (i.e., upper-level) lexical ontology.

Knowledge bases

Knowledge base (KB) is a broader term than ontology. Similar to an ontology, a KB is represented in a KR formalism, which allows automatic inference. It could include multiple axioms, definitions, rules, facts, statements, and any other primitives. In contrast to ontologies, however, KBs are not intended to represent a shared or consensual conceptualization. Thus, ontologies are a specific sort of a KB. Many KBs can be split into ontology and instance data parts, in a way analogous to the splitting of schemata and concrete data in databases.

Proton

PROTON is a light-weight upper-level schema-ontology developed in the scope of the SEKT project, which we will use for ontology-related examples in this section. PROTON is encoded in OWL Lite and defines about 542 entity classes and 183 properties, providing good coverage of named entity types and concrete domains, i.e., modeling of concepts such as people, organizations, locations, numbers, dates, addresses, etc. A snapshot of the PROTON class hierarchy is shown below.



7.4.1.4 Logic and inference

The topics that follow take a closer look at the logic that underlies the retrieval and manipulation of semantic data and the kind of programming that supports it.

Logic programming

Logic programming involves the use of logic for computer programming, where the programmer uses a declarative language to assert statements and a reasoner or theorem-prover is used to solve problems. A reasoner can interpret sentences, such as `IF A THEN B`, as a means to prove `B from A`. In other words, given a collection of logical sentences, a reasoner will explore the solution space in order to find a path to justify the requested theory. For example, to determine the truth value of `C` given the following logical sentences:

```
IF A AND B THEN C
B
IF D THEN A
D
```

a reasoner will interpret the `IF..THEN` statements as rules and determine that `C` is indeed inferred from the KB. This use of rules in logic programming has led to 'rule-based reasoning' and 'logic programming' becoming synonymous, although this is not strictly the case.

In LP, there are rules of logical inference that allow new (implicit) statements to be inferred from other (explicit) statements, with the guarantee that if the explicit statements are true, so are the implicit statements.

Because these rules of inference can be expressed in purely symbolic terms, applying them is the kind of symbol manipulation that can be carried out by a computer. This is what happens when a computer executes a logical program: it uses the rules of inference to derive new statements from the ones given in the program, until it finds one that expresses the solution to the problem that has been formulated. If the statements in the program are true, then so are the statements that the machine derives from them, and the answers it gives will be correct.

The program can give correct answers only if the following two conditions are met:

- The program must contain only true statements;
- The program must contain enough statements to allow solutions to be derived for all the problems that are of interest.

There must also be a reasonable time frame for the entire inference process. To this end, much research has been carried out to determine the complexity classes of various logical formalisms and reasoning strategies. Generally speaking, to reason with Web-scale quantities of data requires a low-complexity approach. A tractable solution is one whose algorithm requires finite time and space to complete.

Predicate logic

From a more abstract viewpoint, the subject of the previous topic is related to the foundation upon which logical programming resides, which is logic, particularly in the form of *predicate logic* (also known as 'first order logic'). Some of the specific features of predicate logic render it very suitable for making inferences over the Semantic Web, namely:

- It provides a high-level language in which knowledge can be expressed in a transparent way and with high expressive power;
- It has a well-understood formal semantics, which assigns unambiguous meaning to logical statements;
- There are proof systems that can automatically derive statements syntactically from a set of premises. These proof systems are both sound (meaning that all derived statements follow semantically from the premises) and complete (all logical consequences of the premises can be derived in the proof system);
- It is possible to trace the proof that leads to a logical consequence. (This is because the proof system is sound and complete.) In this sense, the logic can provide explanations for answers.

The languages of RDF and OWL (Lite and DL) can be viewed as specializations of predicate logic. One reason for such specialized languages to exist is that they provide a syntax that fits well with the intended use (in our case, Web languages based on tags). The other major reason is that they define reasonable subsets of logic. This is important because there is a trade-off between the expressive power and the computational complexity of certain logic: the more expressive the language, the less efficient (in the worst case) the corresponding proof systems. As previously stated, OWL Lite and OWL DL correspond roughly to *description logic*, a subset of predicate logic for which efficient proof systems exist.

Another subset of predicate logic with efficient proof systems comprises the so-called rule systems (also known as *Horn logic* or *definite logic programs*).

A rule has the form:

```
A1, ... , An   B
```

where `Ai` and `B` are atomic formulas. In fact, there are two intuitive ways of reading such a rule:

- If `A1, ... , An` are known to be true, then `B` is also true. Rules with this interpretation are referred to as 'deductive rules'.
- If the conditions `A1, ... , An` are true, then carry out the action `B`. Rules with this interpretation are referred to as 'reactive rules'.

Both approaches have important applications. The deductive approach, however, is more relevant for the purpose of retrieving and managing structured data. This is because it relates better to the possible queries that one can ask, as well as to the appropriate answers and their proofs.

Description logic

Description Logic (DL) has historically evolved from a combination of frame-based systems and predicate logic. Its main purpose is to overcome some of the problems with frame-based systems and to provide a clean and efficient formalism to represent knowledge. The main idea of DL is to describe the world in terms of 'properties' or 'constraints' that specific 'individuals' must satisfy. DL is based on the following basic entities:

- Objects: Correspond to single 'objects' of the real world such as a specific person, a table or a telephone. The main properties of an object are that it can be distinguished from other objects and that it can be referred to by a name. DL objects correspond to the individual constants in predicate logic;
- Concepts: Can be seen as 'classes of objects'. Concepts have two functions: on one hand, they describe *a set of objects* and on the other, they determine *properties* of objects. For example, the class "table" is supposed to describe the set of all table objects in the universe. On the other hand, it also determines some properties of a table such as having legs and a flat horizontal surface that one can lay something on. DL concepts correspond to unary predicates in first order logic and to classes in frame-based systems;
- Roles: Represent relationships between objects. For example, the role 'lays on' might define the relationship between a book and a table, where the book lays upon the table. Roles can also be applied to concepts. However, they do not describe the relationship between the classes (concepts), rather they describe the properties of the objects that are members of that classes;
- Rules: In DL, rules take the form of "if condition *x* (left side), then property *y* (right side)" and form statements that read as "if an object satisfies the condition on the left side, then it has the properties of the right side". So, for example, a rule can state something like 'all objects that are male and have at least one child are fathers'.

The family of DL system consists of many members that differ mainly with respect to the constructs they provide. Not all of the constructs can be found in a single DL system.

7.4.1.5 The Web Ontology Language (OWL) and its dialects

In order to achieve the goal of a broad range of shared ontologies using vocabularies with expressiveness appropriate for each domain, the Semantic Web requires a scalable high-performance storage and reasoning infrastructure. The major challenge towards building such an infrastructure is the expressivity of the underlying standards: RDF, RDFS, OWL, and OWL 2. Even though RDFS can be considered a simple KR language, it is already a challenging task to implement a repository for it, which provides performance and scalability comparable to those of relational database management systems (RDBMS). Even the simplest dialect of OWL (OWL Lite) is a description logic (DL) that does not scale due to reasoning complexity. Furthermore, the semantics of OWL Lite are incompatible with that of RDF(S).

Figure 1 — OWL Layering Map

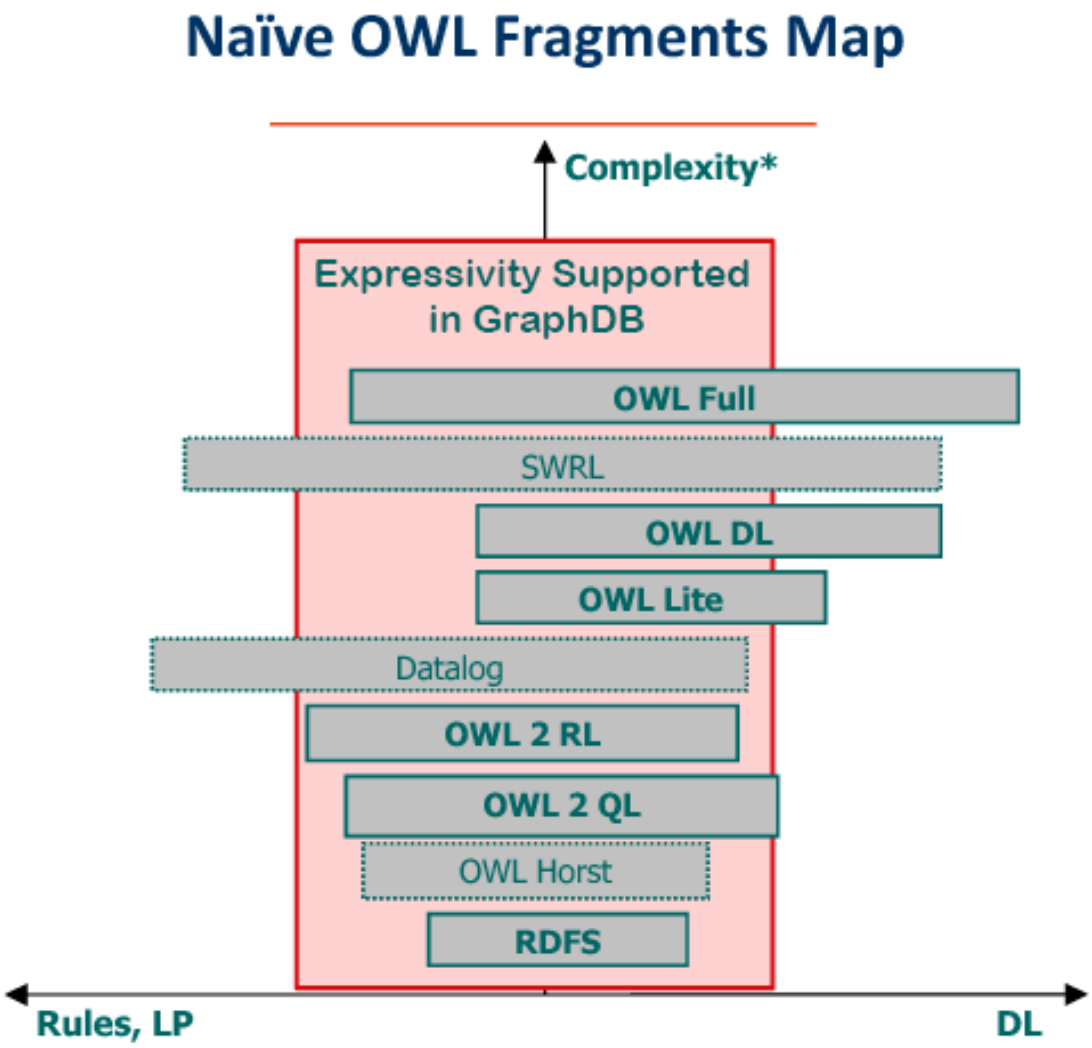
OWL DLP

OWL DLP is a non-standard dialect, offering a promising compromise between expressive power, efficient reasoning, and compatibility. It is defined as the intersection of the expressivity of OWL DL and logic programming. In fact, OWL DLP is defined as the most expressive sub-language of OWL DL, which can be mapped to *Datalog*. OWL DLP is simpler than OWL Lite. The alignment of its semantics to RDFS is easier, as compared to OWL Lite and OWL DL dialects. Still, this can only be achieved through the enforcement of some additional modeling constraints and transformations.

Horn logic and description logic are orthogonal (in the sense that neither of them is a subset of the other). OWL DLP is the 'intersection' of Horn logic and OWL; it is the Horn-definable part of OWL, or stated another way, the OWL-definable part of Horn logic.

DLP has certain advantages:

- From a modeler's perspective, there is freedom to use either OWL or rules (and associated tools and methodologies) for modeling purposes, depending on the modeler's experience and preferences.


Naïve OWL Fragments Map
Complexity*
Expressivity Supported in GraphDB
OWL Full
SWRL
OWL DL
OWL Lite
Datalog
OWL 2 RL
OWL 2 QL
OWL Horst
RDFS
Rules, LP
DL

- From an implementation perspective, either description logic reasoners or deductive rule systems can be used. This feature provides extra flexibility and ensures interoperability with a variety of tools.

Experience with using OWL has shown that existing ontologies frequently use very few constructs outside the DLP language.

OWL-Horst

In "Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity" ter Horst defines RDFS extensions towards rule support and describes a fragment of OWL, more expressive than DLP. He introduces the notion of *R-entailment* of one (target) RDF graph from another (source) RDF graph on the basis of a set of entailment rules *R*. R-entailment is more general than the *D-entailment* used by Hayes in defining the standard RDFS semantics. Each rule has a set of premises, which conjunctively define the body of the rule. The premises are 'extended' RDF statements, where variables can take any of the three positions.

The head of the rule comprises one or more consequences, each of which is, again, an extended RDF statement. The consequences may not contain free variables, i.e., which are not used in the body of the rule. The consequences may contain blank nodes.

The extension of R-entailment (as compared to D-entailment) is that it 'operates' on top of so-called generalized RDF graphs, where blank nodes can appear as predicates. R-entailment rules without premises are used to declare axiomatic statements. Rules without consequences are used to detect inconsistencies.

In this document, we refer to this extension of RDFS as "OWL-Horst". This language has a number of important characteristics:

- It is a proper (backward-compatible) extension of RDFS. In contrast to OWL DLP, it puts no constraints on the RDFS semantics. The widely discussed meta-classes (classes as instances of other classes) are not disallowed in OWL-Horst. It also does not enforce the unique name assumption;
- Unlike DL-based rule languages such as SWRL, R-entailment provides a formalism for rule extensions without DL-related constraints;
- Its complexity is lower than SWRL and other approaches combining DL ontologies with rules.

In Figure 1, the pink box represents the range of expressivity of GraphDB, i.e., including OWL DLP, OWL-Horst, OWL2-RL, most of OWL Lite. However, none of the rulesets include support for the entailment of typed literals (D-entailment).

OWL-Horst is close to what SWAD-Europe has intuitively described as OWL Tiny. The major difference is that OWL Tiny (like the fragment supported by GraphDB) does not support entailment over data types.

OWL2-RL

OWL 2 is a re-work of the OWL language family by the OWL working group. This work includes identifying fragments of the OWL 2 language that have desirable behavior for specific applications/environments.

The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2. This is achieved by defining a syntactic subset of OWL 2, which is amenable to implementation using rule-based technologies, and presenting a partial axiomatization of the OWL 2 RDF-Based Semantics in the form of first-order implications that can be used as the basis for such an implementation. The design of OWL 2 RL was inspired by Description Logic Programs and pD.

OWL Lite

The original OWL specification, now known as OWL 1, provides two specific subsets of OWL Full designed to be of use to implementers and language users. The OWL Lite subset was designed for easy implementation and to offer users a functional subset that provides an easy way to start using OWL.

OWL Lite is a sub-language of OWL DL that supports only a subset of the OWL language constructs. OWL Lite is particularly targeted at tool builders, who want to support OWL, but who want to start with a relatively simple basic set of language features. OWL Lite abides by the same semantic restrictions as OWL DL, allowing reasoning engines to guarantee certain desirable properties.

OWL DL

The OWL DL (where DL stands for Description Logic) subset was designed to support the existing Description Logic business segment and to provide a language subset that has desirable computational properties for reasoning systems.

OWL Full and OWL DL support the same set of OWL language constructs. Their difference lies in the restrictions on the use of some of these features and on the use of RDF features. OWL Full allows free mixing of OWL with RDF Schema and, like RDF Schema, does not enforce a strict separation of classes, properties, individuals and data values. OWL DL puts constraints on mixing with RDF and requires disjointness of classes, properties, individuals and data values. The main reason for having the OWL DL sub-language is that tool builders have developed powerful reasoning systems that support ontologies constrained by the restrictions required for OWL DL.

7.4.1.6 Query languages

In this section, we introduce some query languages for RDF. This may beg the question why we need RDF-specific query languages at all instead of using an XML query language. The answer is that XML is located at a lower level of abstraction than RDF. This fact would lead to complications if we were querying RDF documents with an XML-based language. The RDF query languages explicitly capture the RDF semantics in the language itself.

All the query languages discussed below have a SQL-like syntax, but there are also a few non-SQL-like languages like Versa and Adenine.

The query languages supported by RDF4J (which is the Java framework within which GraphDB operates) and therefore by GraphDB, are SPARQL and SeRQL.

RQL, RDQL

RQL (RDF Query Language) was initially developed by the Institute of Computer Science at Heraklion, Greece, in the context of the European IST project MESMUSES.3. RQL adopts the syntax of OQL (a query language standard for object-oriented databases), and, like OQL, is defined by means of a set of core queries, a set of basic filters, and a way to build new queries through functional composition and iterators.

The core queries are the basic building blocks of RQL, which give access to the RDFS-specific contents of an RDF triplestore. RQL allows queries such as Class (retrieving all classes), Property (retrieving all properties) or Employee (returning all instances of the class with name Employee). This last query, of course, also returns all instances of subclasses of Employee, as these are also instances of the class Employee by virtue of the semantics of RDFS.

RDQL (RDF Data Query Language) is a query language for RDF first developed for Jena models. RDQL is an implementation of the SquishQL RDF query language, which itself is derived from rdfDB. This class of query languages regards RDF as triple data, without schema or ontology information unless explicitly included in the RDF source.

Apart from RDF4J, the following systems currently provide RDQL (all these implementations are known to derive from the original grammar): Jena, RDFStore, PHP XML Classes, 3Store, and RAP (RDF API for PHP).

SPARQL

SPARQL (pronounced "sparkle") is currently the most popular RDF query language; its name is a recursive acronym that stands for "SPARQL Protocol and RDF Query Language". It was standardized by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is now considered a key Semantic Web technology. On 15 January 2008, SPARQL became an official W3C Recommendation.

SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Several SPARQL implementations for multiple programming languages exist at present.

SeRQL

SeRQL (Sesame RDF Query Language, pronounced "circle") is an RDF/RDFS query language developed by Sesame's developer — Aduna — as part of Sesame (now RDF4J). It selectively combines the best features (considered by its creators) of other query languages (RQL, RDQL, N-Triples, N3) and adds some features of its own. As of this writing, SeRQL provides advanced features not yet available in SPARQL. Some of SeRQL's most important features are:

- Graph transformation;
- RDF Schema support;
- XML Schema data-type support;
- Expressive path expression syntax;
- Optional path matching.

7.4.1.7 Reasoning strategies

There are two principle strategies for rule-based inference: Forward-chaining and Backward-chaining:

Forward-chaining to start from the known facts (the explicit statements) and to perform inference in a deductive fashion. Forward-chaining involves applying the inference rules to the known facts (explicit statements) to generate new facts. The rules can then be re-applied to the combination of original facts and inferred facts to produce more new facts. The process is iterative and continues until no new facts can be generated. The goals of such reasoning can have diverse objectives, e.g., to compute the inferred closure, to answer a particular query, to infer a particular sort of knowledge (e.g., the class taxonomy), etc.

Advantages: When all inferences have been computed, query answering can proceed extremely quickly.

Disadvantages: Initialization costs (inference computed at load time) and space/memory usage (especially when the number of inferred facts is very large).

Backward-chaining involves starting with a fact to be proved or a query to be answered. Typically, the reasoner examines the knowledge base to see if the fact to be proved is present and if not it examines the ruleset to see which rules could be used to prove it. For the latter case, a check is made to see what other 'supporting' facts would need to be present to 'fire' these rules. The reasoner searches for proofs of each of these 'supporting' facts in the same way and iteratively maps out a search tree. The process terminates when either all of the leaves of the tree have proofs or no new candidate solutions can be found. Query processing is similar, but only stops when all search paths have been explored. The purpose in query answering is to find not just one but all possible substitutions in the query expression.

Advantages: There are no inferencing costs at start-up and minimal space requirements.

Disadvantages: Inference must be done each and every time a query is answered and for complex search graphs this can be computationally expensive and slow.

As both strategies have advantages and disadvantages, attempts to overcome their weak points have led to the development of various hybrid strategies (involving partial forward- and backward-chaining), which have proven efficient in many contexts.

Total materialization

Imagine a repository that performs total forward-chaining, i.e., it tries to make sure that after each update to the KB, the inferred closure is computed and made available for query evaluation or retrieval. This strategy is generally known as materialization. In order to avoid ambiguity with various partial materialization approaches, let us call such an inference strategy, taken together with the monotonic entailment. When new explicit facts (statements) are added to a KB (repository), new implicit facts will likely be inferred. Under a monotonic logic, adding new explicit statements will never cause previously inferred statements to be retracted. In other words, the addition of new facts can only monotonically extend the inferred closure. Assumption, total materialization.

Advantages and disadvantages of the total materialization:

- Upload/store/addition of new facts is relatively slow, because the repository is extending the inferred closure after each transaction. In fact, all the reasoning is performed during the upload;
- Deletion of facts is also slow, because the repository should remove from the inferred closure all the facts that can no longer be proved;
- The maintenance of the inferred closure usually requires considerable additional space (RAM, disk, or both, depending on the implementation);
- Query and retrieval are fast, because no deduction, satisfiability checking, or other sorts of reasoning are required. The evaluation of queries becomes computationally comparable to the same task for relation database management systems (RDBMS).

Probably the most important advantage of the inductive systems, based on total materialization, is that they can easily benefit from RDBMS-like query optimization techniques, as long as all the data is available at query time. The latter makes it possible for the query evaluation engine to use statistics and other means in order to make 'educated' guesses about the 'cost' and the 'selectivity' of a particular constraint. These optimizations are much more complex in the case of deductive query evaluation.

Total materialization is adopted as the reasoning strategy in a number of popular Semantic Web repositories, including some of the standard configurations of RDF4J and Jena. Based on publicly available evaluation data, it is also the only strategy that allows scalable reasoning in the range of a billion of triples; such results are published by BBN (for DAML DB) and ORACLE (for RDF support in ORACLE 11g).

7.4.1.8 Semantic repositories

Over the last decade, the Semantic Web has emerged as an area where semantic repositories became as important as HTTP servers are today. This perspective boosted the development, under W3C driven community processes, of a number of robust metadata and ontology standards. These standards play the role, which SQL had for the development and spread of the relational DBMS. Although designed for the Semantic Web, these standards face increasing acceptance in areas such as Enterprise Application Integration and Life Sciences.

In this document, the term 'semantic repository' is used to refer to a system for storage, querying, and management of structured data with respect to ontologies. At present, there is no single well-established term for such engines. Weak synonyms are: reasoner, ontology server, metastore, semantic/triple/RDF store, database, repository, knowledge base. The different wording usually reflects a somewhat different approach to implementation, performance, intended application, etc. Introducing the term 'semantic repository' is an attempt to convey the core functionality offered by most of these tools. Semantic repositories can be used as a replacement for database management systems (DBMS), offering easier integration of diverse data and more analytical power. In a nutshell, a semantic repository can dynamically interpret metadata schemata and ontologies, which define the structure and the semantics related to the data and the queries. Compared to the approach taken in a relational DBMS, this allows for easier changing and combining of data schemata and automated interpretation of the data.

7.4.2 Data modeling with RDF(S)

7.4.2.1 What is RDF?

The Resource Description Framework, more commonly known as RDF, is a graph data model that formally describes the semantics, or meaning of information. It also represents metadata, that is, data about data.

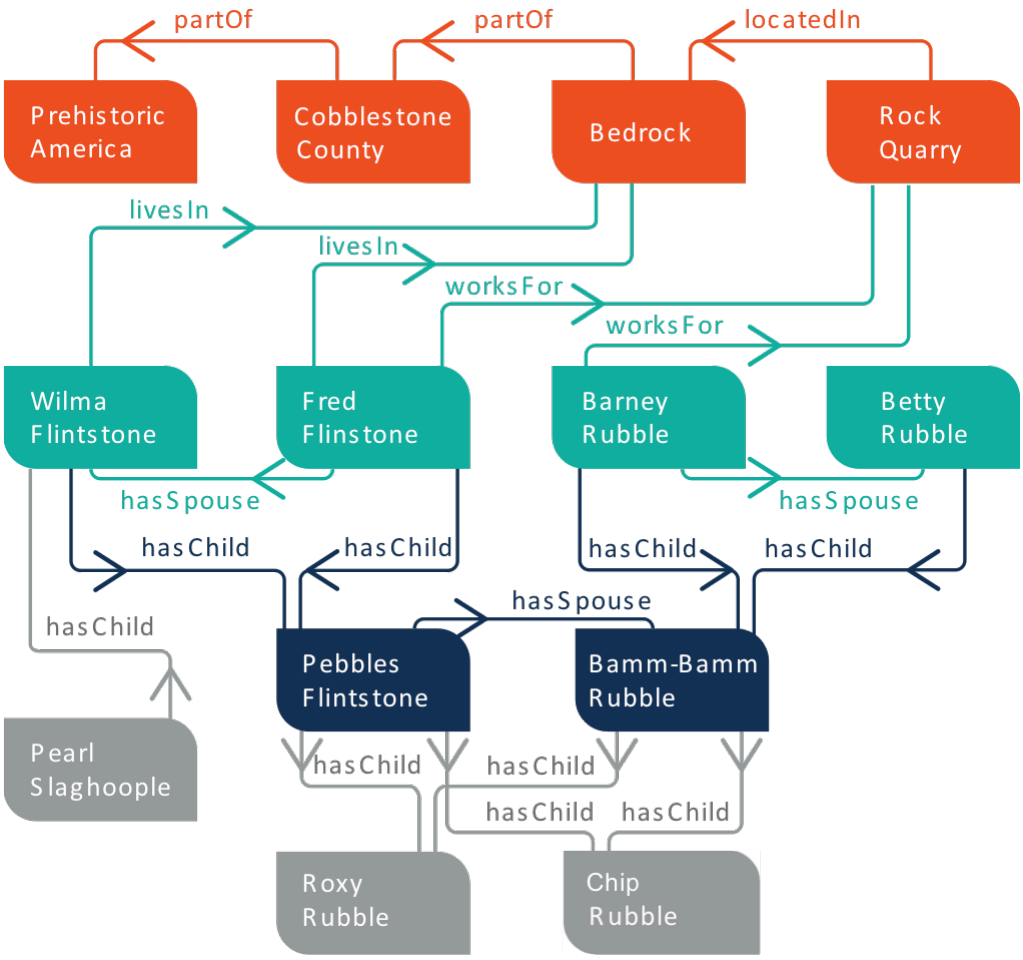
RDF consists of triples. These triples are based on an Entity Attribute Value (EAV) model, in which the subject is the entity, the predicate is the attribute, and the object is the value. Each triple has a unique identifier known as the Uniform Resource Identifier, or URI. URIs look like web page addresses. The parts of a triple, the subject, predicate, and object, represent links in a graph.

Example triples:

subject	predicate	object
:Fred	:hasSpouse	:Wilma
:Fred	:hasAge	25

In the first triple, "Fred hasSpouse Wilma", Fred is the subject, `hasSpouse` is the predicate, and Wilma is the object. Also, in the next triple, "Fred hasAge 25", Fred is the subject, `hasAge` is the predicate and 25 is the object, or value.

Multiple triples link together to form an RDF model. The graph below describes the characters and relationships from the Flintstones television cartoon series. We can easily identify triples such as "WilmaFlintstone livesIn Bedrock" or "FredFlintstone livesIn Bedrock". We now know that the Flintstones live in Bedrock, which is part of Cobblestone County in Prehistoric America.



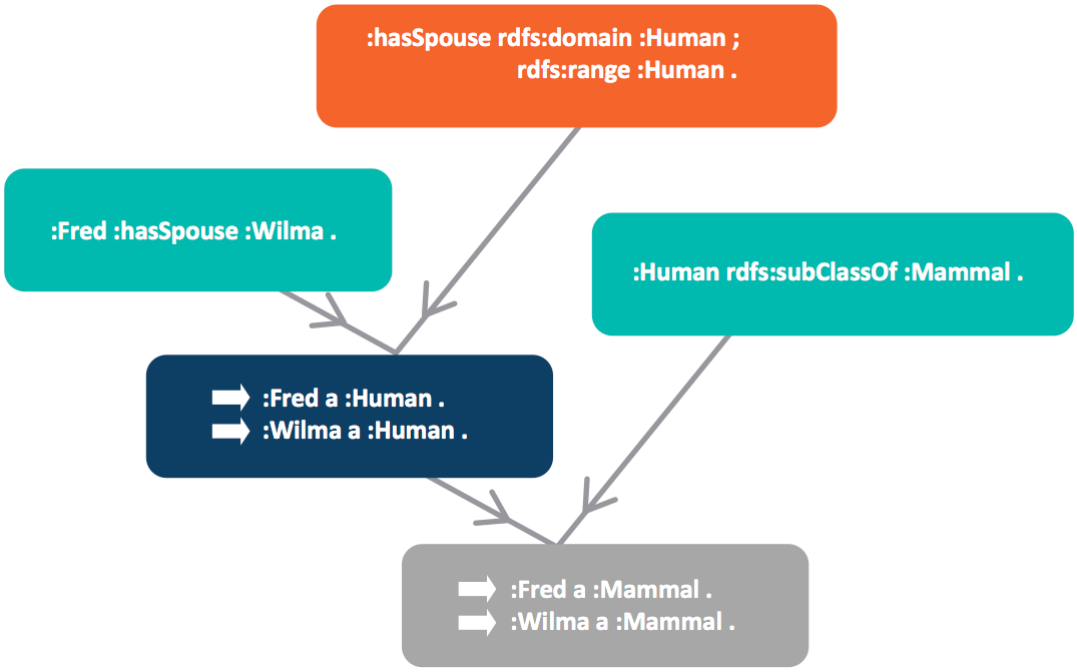
The rest of the triples in the Flintstones graph describe the characters' relations, such as `hasSpouse` or `hasChild`, as well as their occupational association (`worksFor`).

Fred Flintstone is married to Wilma and they have a child Pebbles. Fred works for the Rock Quarry company and Wilma's mother is Pearl Slaghoople. Pebbles Flintstone is married to Bamm-Bamm Rubble who is the child of Barney and Betty Rubble. Thus, as you can see, many triples form an RDF model.

7.4.2.2 What is RDFS?

RDF Schema, more commonly known as RDFS, adds schema to the RDF. It defines a metamodel of concepts like Resource, Literal, Class, and Datatype and relationships such as subClassOf, subPropertyOf, domain, and range. RDFS provides a means for defining the classes, properties, and relationships in an RDF model and organizing these concepts and relationships into hierarchies.

RDFS specifies entailment rules or axioms for the concepts and relationships. These rules can be used to infer new triples, as we show in the following diagram.



Looking at this example, we see how new triples can be inferred by applying RDFS rules to a small RDF/RDFS model. In this model, we use RDFS to define that the `hasSpouse` relationship is restricted to humans. And as you can see, `human` is a subclass of `mammal`.

If we assert that Wilma is Fred's spouse using the `hasSpouse` relationship, then we can infer that Fred and Wilma are human because, in RDFS, the `hasSpouse` relationship is defined to be between humans. Because we also know humans are mammals, we can further infer that Fred and Wilma are mammals.

7.4.3 The SPARQL query language

7.4.3.1 What is SPARQL?

SPARQL is a SQL-like query language for RDF data. SPARQL queries can produce result sets that are tabular or RDF graphs depending on the kind of query used.

- `SELECT` is similar to the SQL SELECT in that it produces tabular result sets.
- `CONSTRUCT` creates a new RDF graph based on query results.
- `ASK` returns Yes or No depending on whether the query has a solution.
- `DESCRIBE` returns the RDF graph data about a resource. This is, of course, useful when the query client does not know the structure of the RDF data in the data source.
- `INSERT` adds triples to a graph,

- `DELETE` removes triples from a graph.

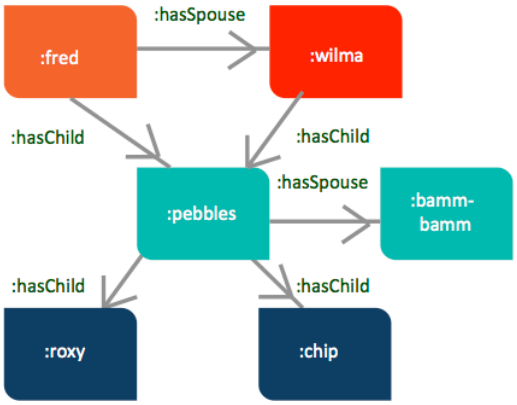
Let's use SPARQL, the query language for RDF graphs, to create a graph. To write the SPARQL query that creates an RDF graph, perform these steps:

First, define prefixes to URIs with the `PREFIX` keyword. In the example below, we set `bedrock` as the default namespace for the query.

Next, use `INSERT DATA` to signify you want to insert statements. Write the subject predicate object statements.

Finally, execute this query:

```
PREFIX : <http://bedrock/>
INSERT DATA {
  :fred     :hasSpouse  :wilma .
  :fred     :hasChild   :pebbles .
  :wilma    :hasChild   :pebbles .
  :pebbles  :hasSpouse  :bamm-bamm ;
            :hasChild   :roxy, :chip .
}
```



As you can see in the example shown in the gray box, we wrote a query which included `PREFIX`, `INSERT DATA`, and several subject predicate object statements, which are:

Fred has spouse Wilma, Fred has child Pebbles, Wilma has child Pebbles, Pebbles has spouse Bamm-Bamm, and Pebbles has children Roxy and Chip.

Now, let's write a SPARQL query to access the RDF graph you just created.

First, define prefixes to URIs with the `PREFIX` keyword. As in the earlier example, we set `bedrock` as the default namespace for the query.

Next, use `SELECT` to signify you want to select certain information, and `WHERE` to signify your conditions, restrictions, and filters.

Finally, execute this query:

```
PREFIX : <http://bedrock/>
SELECT ?subject ?predicate ?object
WHERE {?subject ?predicate ?object }
```

subject	predicate	object
:fred	:hasChild	:pebbles
:pebbles	:hasChild	:roxy
:pebbles	:hasChild	:chip
:wilma	:hasChild	:pebbles

As you can see in this example shown in the gray box, we wrote a SPARQL query which included `PREFIX`, `SELECT`, and `WHERE`. The red box displays the information which is returned in response to the written query. We can see the familial relationships between Fred, Pebbles, Wilma, Roxy, and Chip.

SPARQL is quite similar to SQL, however, unlike SQL which requires SQL schema and data in SQL tables, SPARQL can be used on graphs and does not need a schema to be defined initially.

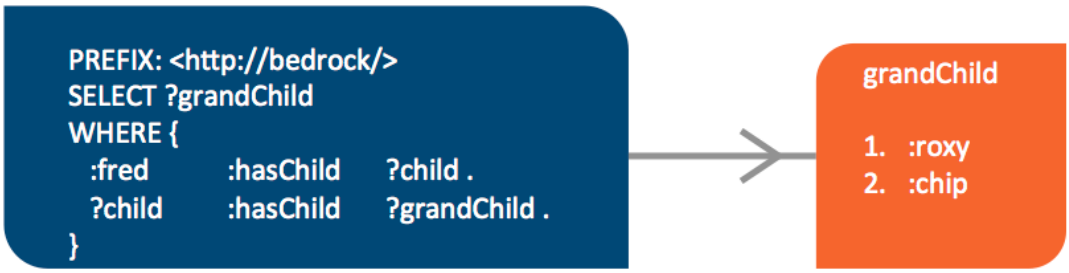
In the following example, we will use SPARQL to find out if Fred has any grandchildren.

First, define prefixes to URIs with the `PREFIX` keyword.

Next, we use `ASK` to discover whether Fred has a grandchild, and `WHERE` to signify the conditions.

As you can see in the query in the green box, Fred's children's children are his grandchildren. Thus the query is easily written in SPARQL by matching Fred's children and then matching his children's children. The `ASK` query returns "Yes" so we know Fred has grandchildren.

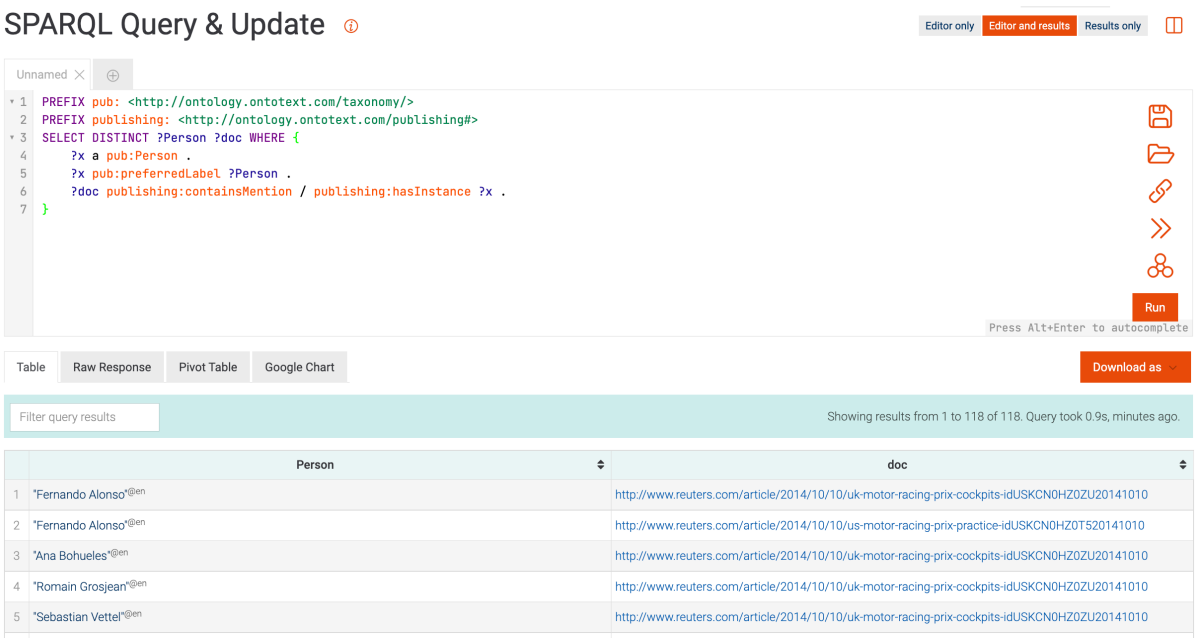
If instead we want a list of Fred's grandchildren we can change the `ASK` query to a `SELECT` one:



The query results, reflected in the red box, tell us that Fred's grandchildren are Roxy and Chip.

7.4.3.2 Using SPARQL in GraphDB

The easiest way to execute SPARQL queries in GraphDB is by using the GraphDB Workbench. Just choose *SPARQL* from the navigation bar, enter your query and hit *Run*, as shown in this example:



7.4.4 Create a custom graph view over your RDF data

RDF is the most popular format for exchanging semantic data. Unlike logical database models, ontologies are optimized to correctly represent the knowledge in a particular business domain. This means that their structure is often verbose, includes abstract entities to express OWL axioms, and contains implicit statements and complex N-ary relationship with provenance information. Graph View is a user interface optimized for mapping knowledge base models to simpler edge and vertex models configured by a list of SPARQL queries.

7.4.4.1 How does it work?

The Graph View interface accepts four different SPARQL queries to retrieve data from the knowledge base:

- Node expansion determines how new nodes and links are added to the visual graph when the user expands an existing node.
- Node type, size, and label control the node appearance. Types correspond to different colors. Each binding is optional.
- Vertex (i.e., predicate) label determines where to read the name.
- Node info controls all data visible for the resource displayed in tabular format. If an `?image` binding is found in the results, the value is used as an image source.

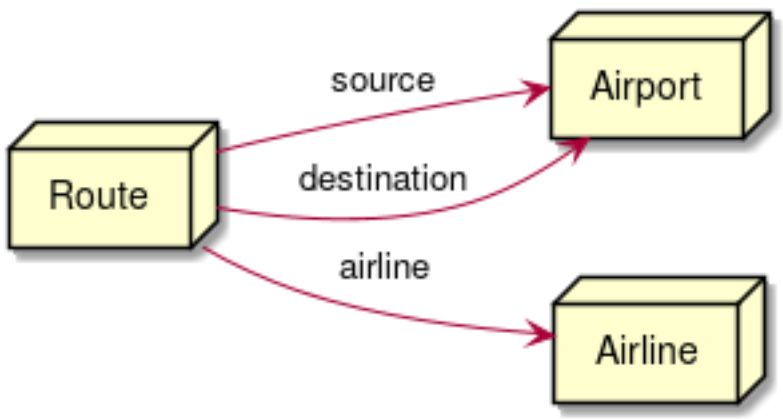
By using these four queries, you may override the default configuration and adapt the knowledge base visualization to:

- Integrate custom ontology schema and the preferred label;
- Hide provenance or another metadata related information;
- Combine nodes, so you can skip relation objects and show them as a direct link;
- Filter instances with all sorts of tools offered by the SPARQL language;
- Generate RDF resources on the fly from existing literals.

7.4.4.2 World airport, airline, and route data

The OpenFlights Airports Database contains over 10,000 airports, train stations, and ferry terminals spanning the globe. Airport base data was generated by from DAFIF (October 2006 cycle) and OurAirports, plus time zone information from EarthTools. All DST information are added manually. Significant revisions and additions have been made by the users of OpenFlights. Airline data was extracted directly from Wikipedia's gargantuan List of airlines. The dataset can easily link to DBpedia and be integrated with the rest of linked open data cloud.

Data model

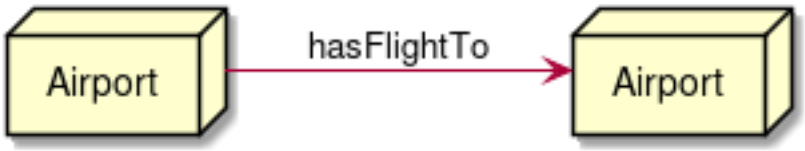


All OpenFlight CSV files are converted by using Ontotext Refine. To start exploring, first import the `airports.ttl` dataset which contains the data in RDF.

Configured queries

Find how airports are connected with flights

Let's find out how the airports are connected by skipping the route relation and model a new relation `hasFlightTo`.



In SPARQL, this can be done with the following query:

```
PREFIX onto: <http://www.ontotext.com/>
construct {
  ?source onto:hasFlightTo ?destination .
}  where {
    ?flight <http://openflights.org/resource/route/sourceId> ?source .
    ?flight <http://openflights.org/resource/route/destinationId> ?destination
}
```

Using the *Visual* button in the SPARQL editor, we can see the results of this query as a visual graph.

We can also save the graph and expand to more airports. To do this, navigate to *Explore* ‣ *Visual graph* and click *Create graph config*.

Note: Any visual graph that a particular user saves will be available with all repositories that that user has access to.

First you are asked to select the initial state of your graph. For simplicity, we choose to start with a query and enter from above. Now let's make this graph expandable by configuring the Graph expansion query:

```
PREFIX onto: <http://www.ontotext.com/>
construct {
  ?node onto:hasFlightTo ?destination .
}  where {
    ?flight <http://openflights.org/resource/route/sourceId> ?node .
    ?flight <http://openflights.org/resource/route/destinationId> ?destination .
} limit 100
```

You can also select a different airport to start from every time by making the starting point a search box.

Find which airlines fly to which airports

The power of the visual graph is that we can create multiple Graph views on top of the same data. Let's create a new one using the following query:

```
PREFIX onto: <http://www.ontotext.com/>
construct {
  ?airport onto:hasFlightFromWithAirline ?airline .
}  where {
    ?route <http://openflights.org/resource/route/sourceId> ?airport .
```

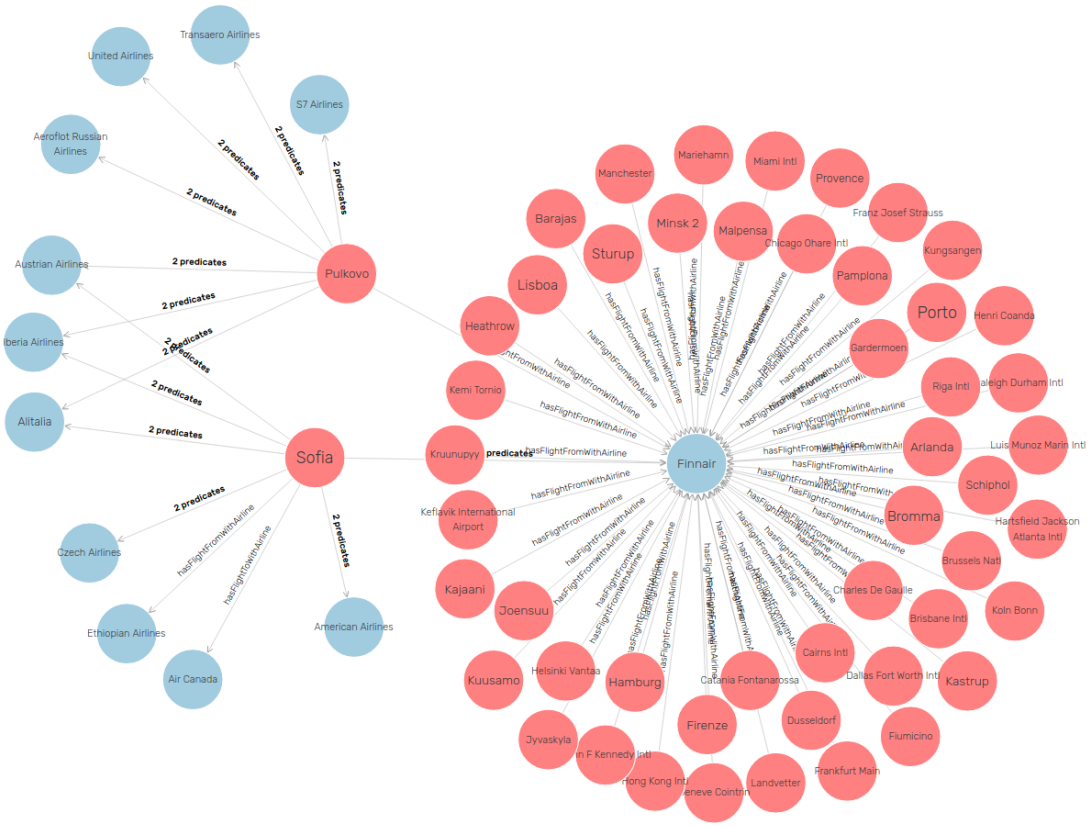
(continues on next page)

(continued from previous page)

```
    ?route <http://openflights.org/resource/route/airlineId> ?airline .
} limit 100
```

And let's create a visual graph with the following expand query:

```
# Note that ?node is the node you clicked and must be used in the query
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
PREFIX onto: <http://www.ontotext.com/>

CONSTRUCT {
    # The triples that will be added to the visual graph when you expand airports
    ?node onto:hasFlightFromWithAirline ?airline1 .
    ?node onto:hasFlightToWithAirline ?airline2 .

    # The triples to be added when you expand airlines
    ?airport1 onto:hasFlightFromWithAirline ?node .
    ?airport2 onto:hasFlightToWithAirline ?node .
} WHERE {
    {
        # Incoming flights for airport
        ?route <http://openflights.org/resource/route/sourceId> ?node .
        ?route <http://openflights.org/resource/route/airlineId> ?airline1 .

    } UNION {
        # Outgoing flights for airport
        ?route <http://openflights.org/resource/route/destinationId> ?node .
        ?route <http://openflights.org/resource/route/airlineId> ?airline2 .
    } UNION
    {
        # Incoming flights for airline
        ?route <http://openflights.org/resource/route/sourceId> ?airport1 .
        ?route <http://openflights.org/resource/route/airlineId> ?node .

    } UNION {
        # Outgoing flights for airline
        ?route <http://openflights.org/resource/route/destinationId> ?airport2 .
        ?route <http://openflights.org/resource/route/airlineId> ?node .
    }
}
```

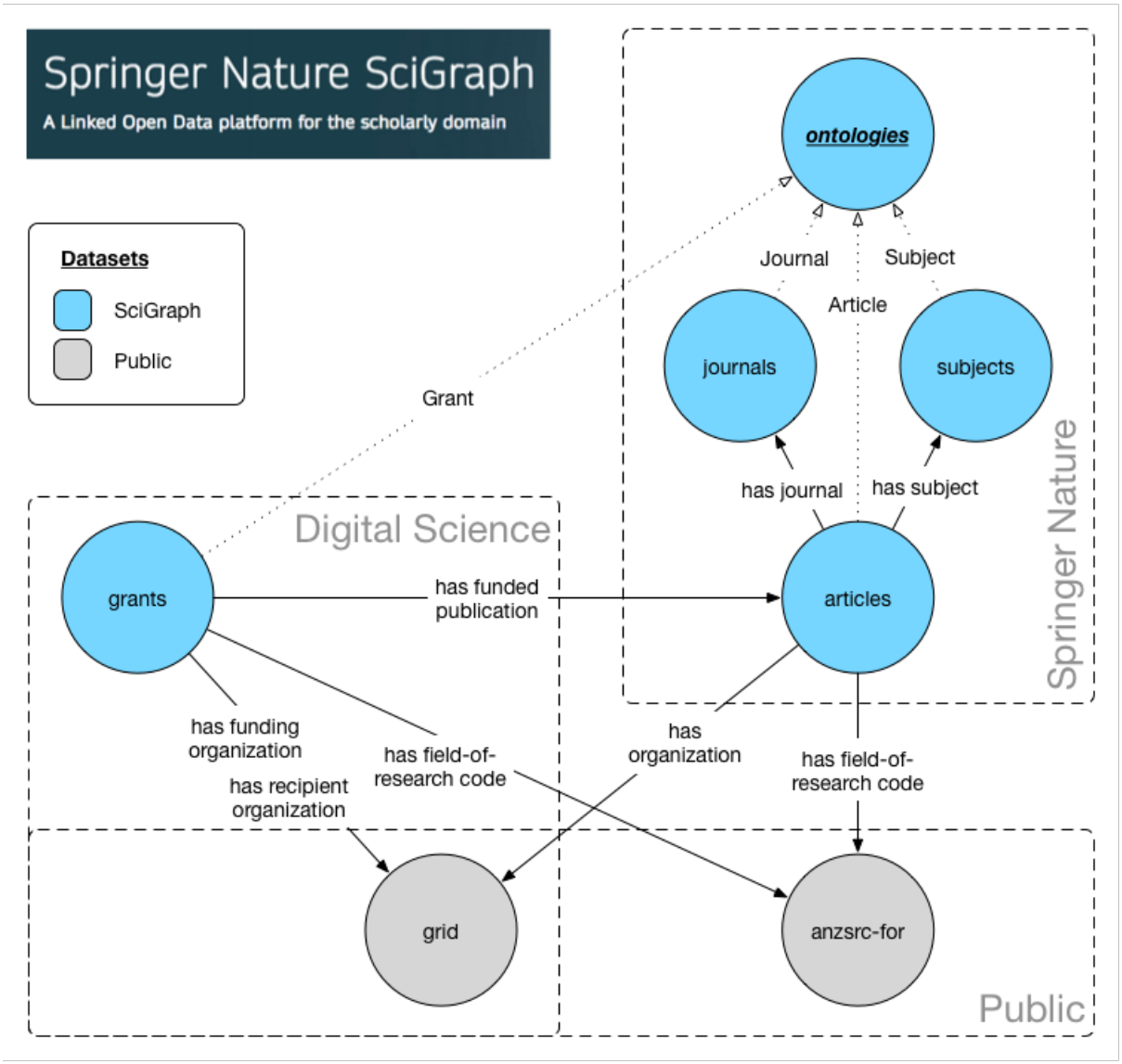


7.4.4.3 Springer Nature SciGraph

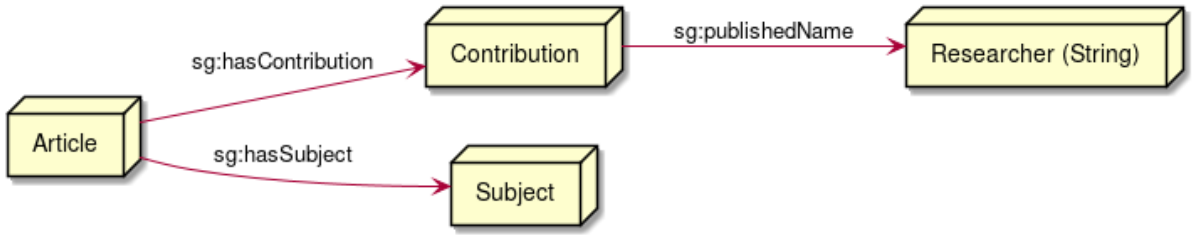
SciGraph is a Linked Open Data platform for the scholarly domain. The dataset aggregates data sources from Springer Nature and key partners from the domain. It collates information from across the research landscape, such as funders, research projects, conferences, affiliations, and publications.

Data model

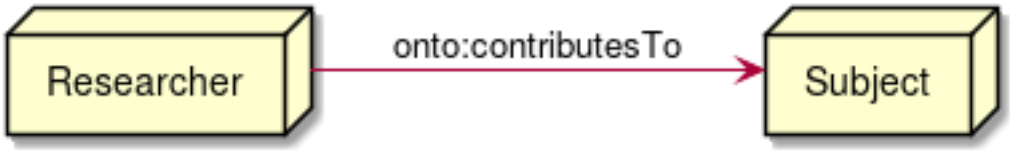
This is the full data model:



but let's say we are only interested in articles, contributions, and subjects.

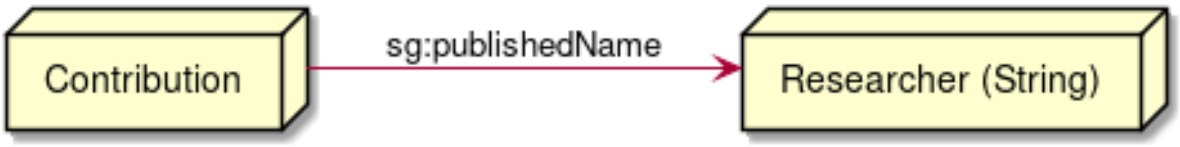


From this we can say that a researcher contributes to a subject, and create a virtual URI for the researcher since it is a Literal.



Find researchers that contribute to the same subjects

We do not have a URI for a researcher. How can we search for researchers?



Navigate to *Setup* ▸ *Autocomplete* and add a `sg:publishedName` predicate. The retrieved result will be contributions by given names in the search box.

Now let's create the graph config. We need to configure an expansion for contribution since this is our starting point for both subjects and researchers.

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX onto: <http://www.ontotext.com/>
PREFIX grid: <http://www.grid.ac/ontology/>
construct {
     ?node onto:publishedNameURI ?researcherNameUri1 .
     ?node onto:isContributorFor ?subject .
     ?researcherNameUri2 onto:isContributorFor ?node .
}
where {
    #BIND (onto:Declan_Butler as ?node)
    #BIND (<http://www.springernature.com/scigraph/things/subjects/policy> as ?node)

    {
        BIND( IRI(CONCAT("http://www.ontotext.com/", REPLACE(STR(?researcherName)," ","_"))) as ?
↪researcherNameUri1)
        ?node sg:publishedName ?researcherName .
    } UNION {
        BIND( REPLACE(REPLACE(STR(?node),"_"," "), "http://www.ontotext.com/" , "") as ?researcherName)
        ?contribution a sg:Contribution .
        ?contribution sg:publishedName ?researcherName .
        ?article sg:hasContribution ?contribution .
        ?article sg:hasSubject ?subject .
    }
    UNION {
        BIND( IRI(CONCAT("http://www.ontotext.com/", REPLACE(STR(?researcherName)," ","_"))) as ?
↪researcherNameUri2)
        ?contribution a sg:Contribution .
        ?contribution sg:publishedName ?researcherName .
        ?article sg:hasContribution ?contribution .
        ?article sg:hasSubject ?node .
    }
}
```

However, not all researchers have contributions to articles with subjects. Let's use an initial query that will fetch some researchers that have such relations. This is just a simplified version of the query above fetching some researchers and subjects.

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX onto: <http://www.ontotext.com/>
PREFIX grid: <http://www.grid.ac/ontology/>
construct {
    ?researcherNameUri2 onto:isContributorFor ?node .
}
where {
```

(continues on next page)

(continued from previous page)

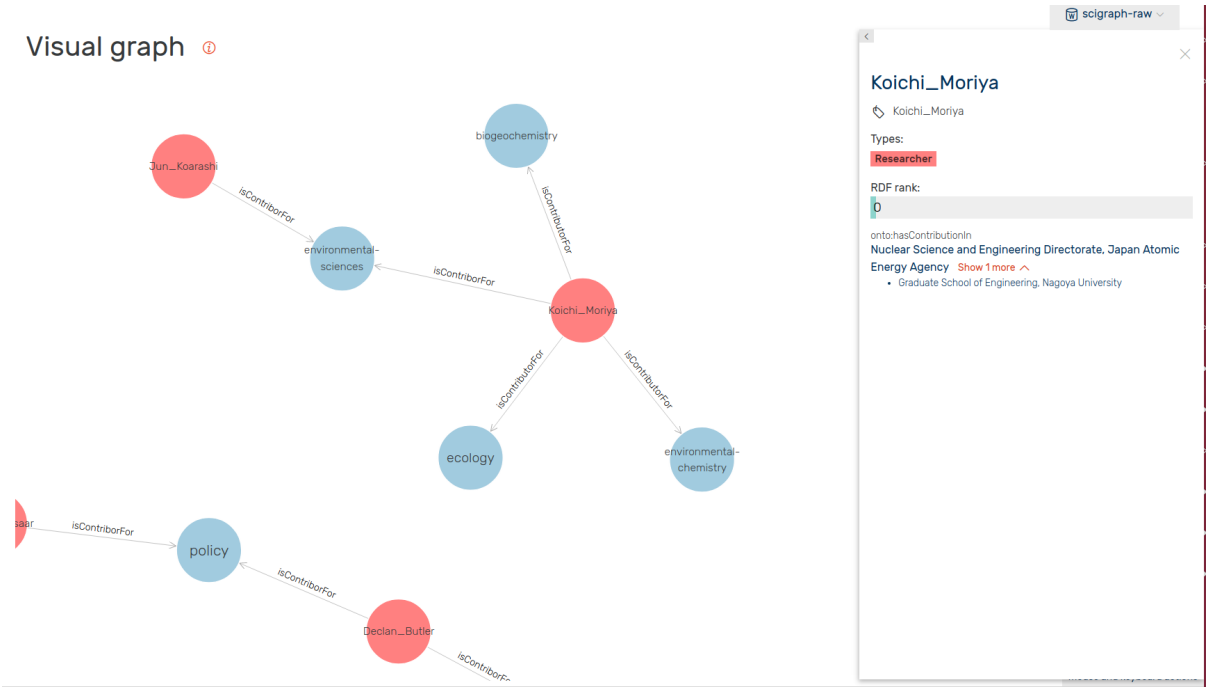
```
 {
        BIND( IRI(CONCAT("http://www.ontotext.com/", REPLACE(STR(?researcherName)," ","_"))) as ?
↪researcherNameUri2)
        ?contribution a sg:Contribution .
        ?contribution sg:publishedName ?researcherName .
        ?article sg:hasContribution ?contribution .
        ?article sg:hasSubject ?node .
    }
} limit 100
```

But the nodes in our graph are all the same since they do not have RDF types. Now let's configure the way the types of the nodes are obtained.

```
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>

SELECT distinct ?type  {
  #  BIND (<http://www.ontotext.com/S._R._Arnold> as ?node)
#    # Get node type
    OPTIONAL {?node ?p ?o}
    BIND(IF (strStarts(STR(?node), "http://www.ontotext.com/"), "Researcher", "Subject") as ?type)


} ORDER BY ?type
```

But what if we want to see additional data for each node, i.e., which university has a researcher contribution for:

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
SELECT distinct ?property ?value where {
#    BIND (<http://www.ontotext.com/Kevin_J._Gaston> as ?node)
    BIND (<http://www.ontotext.com/hasContributionIn> as ?property)
    BIND( REPLACE(REPLACE(STR(?node),"_"," "), "http://www.ontotext.com/" , "") as ?researcherName)
    optional {?node ?p ?o}
    ?contribution sg:publishedName ?researcherName .
    ?contribution sg:hasAffiliation ?affiliation .
    ?affiliation sg:publishedName ?value .
} limit 100
```



7.4.5 RDF-star and SPARQL-star

7.4.5.1 The modeling challenge

RDF is an abstract knowledge representation model that does not differentiate data from metadata. This prevents the extension of an existing model with statement-level metadata annotations like certainty scores, weights, temporal restrictions, and provenance information like if this was a manually modified annotation. Several approaches discussed on this page mitigate the inherent lack of native support for such annotations in RDF. However, they all have certain advantages and disadvantages, which we will look at below.

Standard reification

Reification means expressing an abstract construct with the existing concrete methods supported by the language. The RDF specification sets a standard vocabulary for representing references to statements like:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix voc: <http://example.com/voc#> .

voc:man voc:hasSpouse voc:woman .
voc:id1 rdf:type rdf:Statement ;
    rdf:subject voc:man ;
    rdf:predicate voc:hasSpouse ;
    rdf:object voc:woman ;
    voc:startDate "2020-02-11"^^xsd:date .
```

Standard reification requires stating four additional triples to refer to the triple for which we want to provide metadata. The subject of these four additional triples has to be a new identifier (IRI or blank node), which later on may be used for providing the metadata. The existence of a reference to a triple does not automatically assert it. The main advantage of this method is the standard support by every RDF store. Its disadvantages are the inefficiency related to exchanging or persisting the RDF data and the cumbersome syntax to access and match the corresponding four reification triples.

N-ary relations

The approach for representing N-ary relations in RDF is to model it via a new relationship concept that connects all arguments like:

```
@prefix voc: <http://example.com/voc#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

voc:Marriage1 rdf:type voc:Marriage ;
    voc:partner1 voc:man ;
    voc:partner2 voc:woman ;
    voc:startDate "2020-02-11"^^xsd:date .
```

The approach is similar to standard reification, but it adopts a schema specific to the domain model that is presumably understood by its consumers. The only disadvantage here is that this approach increases the ontology model complexity and is proven difficult to evolve models in a backward compatible way.

RDF-star and SPARQL-star

RDF-star (formerly RDF*) is an extension of the RDF 1.1 standard that proposes a more efficient reification serialization syntax. The main advantages of this representation include reduced document size that increases the efficiency of data exchange, as well as shorter SPARQL queries for improved comprehensibility.

```
@prefix voc: <http://example.com/voc#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

voc:man voc:hasSpouse voc:woman .
<<voc:man voc:hasSpouse voc:woman>> voc:startDate "2020-02-11"^^xsd:date .
```

The RDF-star extension captures the notion of an *embedded triple* by enclosing the referenced triple using the strings `<<` and `>>`. The embedded triples, like the blank nodes, may take a subject and object position only, and their meaning is aligned to the semantics of the standard reification, but using a much more efficient serialization syntax. To simplify the querying of the embedded triples, the paper extends the query syntax with SPARQL-star (formerly SPARQL*) enabling queries like:

```
# List all metadata for the given reference to a statement
PREFIX voc: <http://example.com/voc#>
SELECT *
WHERE {
    <<voc:man voc:hasSpouse voc:woman>> ?p ?o
}
```

The embedded triple in SPARQL-star also supports free variables for retrieving a list of reference statements:

```
# List all metadata for the given reference to a statement
PREFIX voc: <http://example.com/voc#>
SELECT *
WHERE {
    <<?man voc:hasSpouse voc:woman>> ?p ?o
    FILTER (?man = voc:man)
}
```

7.4.5.2 How do the different approaches compare?

To test the different approaches, we benchmark a subset of Wikidata, whose data model heavily uses statement-level metadata. The authors of the paper Reifying RDF: What works well with Wikidata? have done an excellent job with remodeling the dataset in various formats, and kindly shared with our team the output datasets. According to their modeling approach, the dataset includes:

Modeling approach	Total statements	Loading time (min)	Repository image size (MB)
Standard reification	391,652,270	52.4	36,768
N-ary relations	334,571,877	50.6	34,519
Named graphs	277,478,521	56	35,146
RDF-star	220,375,702	34	22,465

We did not test the singleton properties approach due to the high number of unique predicates.

7.4.5.3 Syntax and examples

The section provides more in-depth details on how GraphDB implements the RDF-star/SPARQL-star syntax. Let's say we have a statement like the one above, together with the metadata fact that we are 90% certain about this statement. The RDF-star syntax allows us to represent both the data and the metadata by using an embedded triple as follows:

```
@prefix ex:  <http://example.com/ns#> .
@prefix voc: <http://example.com/voc#> .

<<voc:man voc:hasSpouse voc:woman>> ex:certainty 0.9 .
```

According to the formal semantics of RDF-star, each embedded triple also asserts the referenced statement and its retraction — deletes it. Unfortunately, this requirement breaks the compatibility with the standard reification and causes a non-transparent behavior when dealing with triples stored in multiple named graphs. GraphDB implements the embedded triples by introducing a new additional RDF type next to IRI, blank node, and literal. So in the previous example, the engine will store only a single triple.

> **Warning:** GraphDB will not explicitly assert the referenced statement by an embedded triple! Every embedded triple acts as a new RDF type, which means only a reference to a statement.

Below are a few more examples of how this syntax can be used. All assume that *voc:* and *xsd:* have been declared as in the above examples.

- Object relation qualifiers:

  ```
  <<voc:man voc:hasSpouse voc:woman>> voc:startDate "2020-02-11"^^xsd:date
  ```

  `voc:hasSpouse` is a symmetric relation so that it can be inferred in the opposite direction. However, the metadata in the opposite direction is not asserted automatically, so it needs to be added:

  ```
  <<voc:woman voc:hasSpouse voc:man>> voc:startDate "2020-02-11"^^xsd:date
  ```

- Data value qualifiers:

  ```
  <<voc:painting voc:height 32.1>>
    voc:unit voc:cm;
    voc:measurementTechnique voc:laserScanning;
    voc:measuredOn "2020-02-11"^^xsd:date.
  ```

- Statement sources/references:

  ```
  <<voc:man voc:hasSpouse voc:woman>>
    voc:source voc:TheNationalEnquirer;
    voc:webpage <http://nationalenquirer.com/news/2020-02-12>;
    voc:retrieved "2020-02-13"^^xsd:date.
  ```

- Nested embedded triples:

  ```
  << <<voc:man voc:hasSpouse voc:woman>> voc:startDate "2020-02-11"^^xsd:date >>
      voc:webpage <http://nationalenquirer.com/news/2020-02-12> .
  ```

Carried over into the syntax of the extended query language SPARQL-star, triple patterns can be embedded as well. This provides a query syntax in which accessing specific metadata about a triple is just a matter of mentioning the triple in the subject or object position of a metadata-related triple pattern. For example, by adopting the aforementioned syntax for nesting, we can query for all age statements and their respective certainty as follows:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex:   <http://example.com/ns#>
```

(continues on next page)

(continued from previous page)

```
SELECT ?p ?a ?c WHERE {
    <<?p foaf:age ?a>> ex:certainty ?c .
}
```

Additionally, SPARQL-star modifies the **BIND** clauses to select a group of embedded triples by using free variables:

```
PREFIX ex:   <http://example.com/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?p ?a ?c WHERE {
    BIND (<<?p foaf:age ?a>> AS ?t)
    ?t ex:certainty ?c .
}
```

The semantics of **BIND** has a deviation from that of the other RDF types. When binding an embedded triple, it creates an iterator over the triple entities that match its components and binds these to the target variable. As a result, the **BIND**, when used with three constants, works like a **FILTER**. The same does not apply for **VALUES**, which will return any value.

```
PREFIX ex:   <http://example.com/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {

    {
        # Binds the value to ?literal variable
        BIND ("new value for the store" as ?literal)
    }
    UNION
    {
        # Returns empty value and acts like a FILTER
        BIND (<<ex:subject foaf:name "new value for the store">> AS ?triple)
    }
    UNION
    {
        # Values generates new values
        VALUES ?newTriple { <<ex:subject foaf:name "new value for the store">> }
    }
}
```

To avoid any parsing of the embedded triple, GraphDB introduces multiple new SPARQL functions:

```
PREFIX voc: <http://example.com/voc#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {

    VALUES ?triple { <<voc:man voc:hasSpouse voc:woman>> }

    # Checks if the variable is of type embedded triple
    BIND (rdf:isTriple(?triple) as ?isTriple)

    # Extract the subject, predicate or object from an embedded triple
    BIND (rdf:subject(?triple) as ?subject)
    BIND (rdf:predicate(?triple) as ?predicate)
    BIND (rdf:object(?triple) as ?object)

    # Create a new embedded statement
    BIND (rdf:Statement(?subject, ?predicate, ?object) as ?newTriple)
```

(continues on next page)

(continued from previous page)

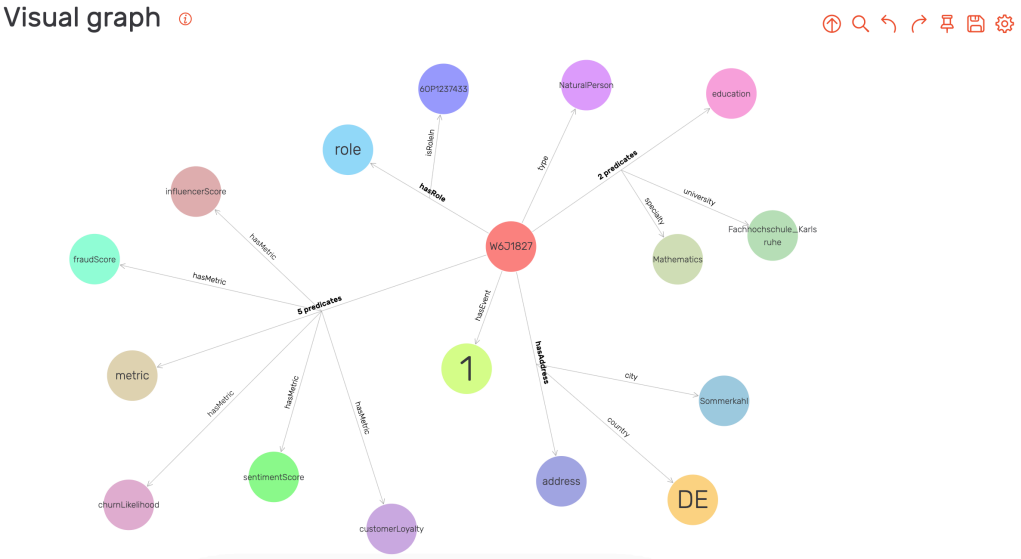
```
}
```

This also showcases the fact that in SPARQL-star, variables in query results may be bound not only to IRIs, literals, or blank nodes, but also to full RDF-star triples.

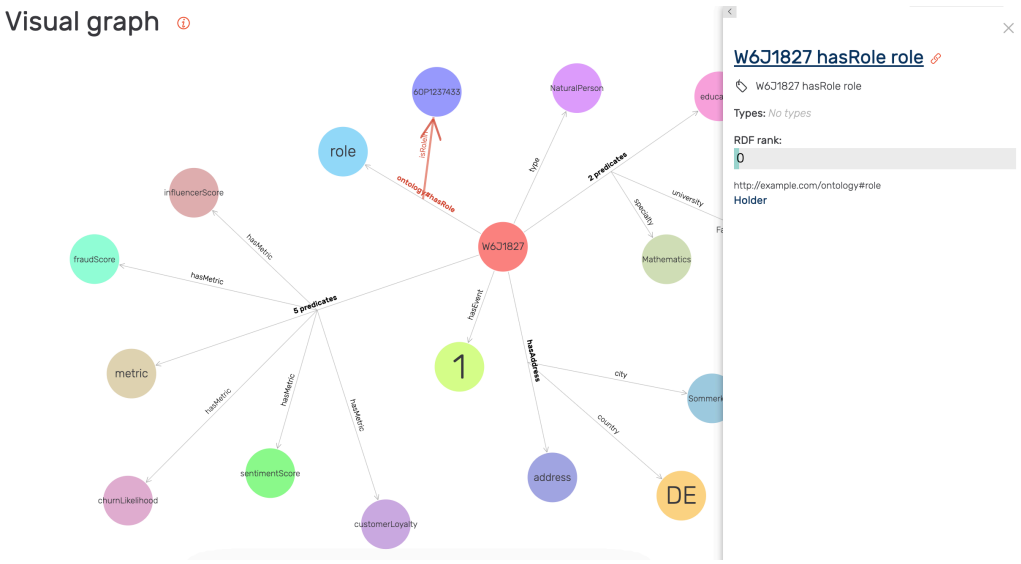
Embedded triple visualization

We can also visualize embedded triples in GraphDB's Visual graph.

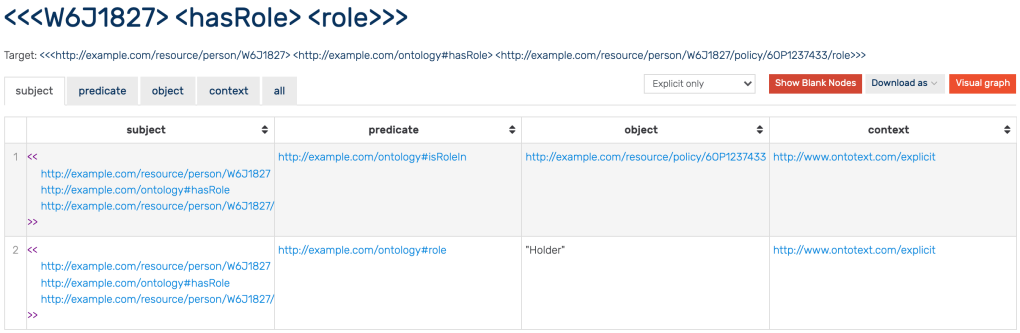
1. Download this small `dataset` with some Wikidata information about a person.
2. Upload it into a GraphBD repository.
3. Enable autocompletion from *Setup* ▸ *Autocomplete*.
4. Go to *Explore* ▸ *Visual graph* and look up the resource `W6J1827`.
5. The following visualization with embedded triples will be displayed. Note that the predicate labels of the embedded triples are bolded.



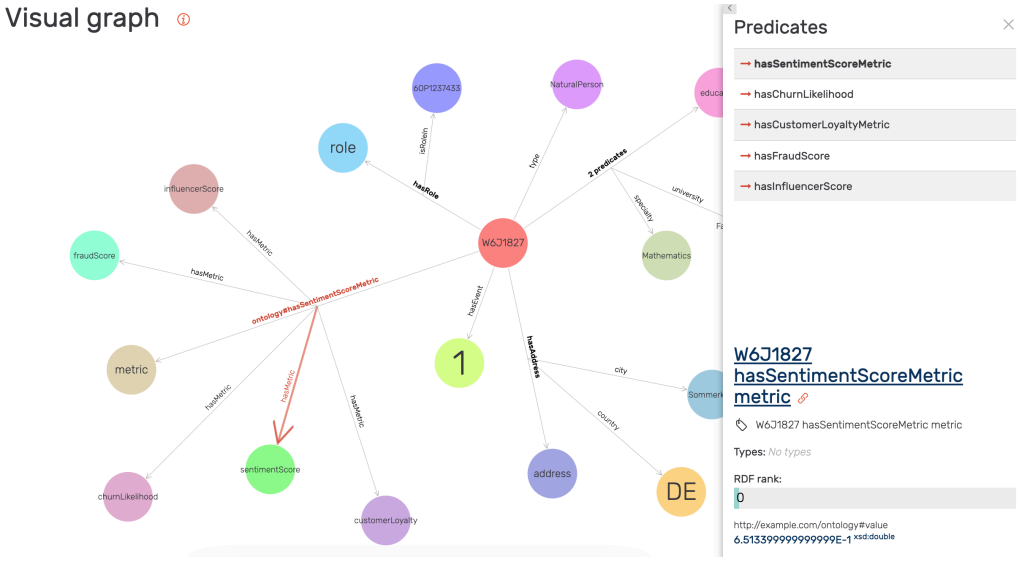
6. When the embedded triple contains just one link, click on the predicate label to explore it:



The edge will be highlighted, and in the side panel that opens you can view more details about the predicate. You can also click on it to open it in the resource view.



7. When the embedded triple we want to explore contains more than one link, click on its predicate label to see a list with all of the embedded predicates in the side panel. Click on an embedded predicate to view more details about it.



7.4.5.4 Convert standard reification to RDF-star

The RDF-star support in GraphDB does not exclude any of the other modeling approaches. It is possible to independently maintain RDF-star and standard reification statements in the same repository, like:

```
@prefix voc: <http://example.com/voc#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

voc:man voc:hasSpouse voc:woman .
voc:id1 rdf:type rdf:Statement ;
    rdf:subject voc:man ;
    rdf:predicate voc:hasSpouse ;
    rdf:object voc:woman ;
    voc:startDate "2020-02-11"^^xsd:date .

<<voc:man voc:hasSpouse voc:woman>> voc:startDate "2020-02-11"^^xsd:date .
```

Still, this is likely to confuse, so GraphDB provides a tool for converting standard reification to RDF-star outside of the database using the *reification-convert* command line tool. If the data is already imported, use this SPARQL for a conversion:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

DELETE {
    ?reification a rdf:Statement .
    ?reification rdf:subject ?subject .
    ?reification rdf:predicate ?predicate .
    ?reification rdf:object ?object .
    ?reification ?p ?o .
} INSERT {
    <<?subject ?predicate ?object>> ?p ?o .
} WHERE {
    ?reification a rdf:Statement .
    ?reification rdf:subject ?subject .
    ?reification rdf:predicate ?predicate .
    ?reification rdf:object ?object .
    ?reification ?p ?o .
    FILTER (?p NOT IN (rdf:subject, rdf:predicate, rdf:object) &&
    (?p != rdf:type && ?object != rdf:Statement))
}
```

7.4.5.5 MIME types and file extensions for RDF-star in RDF4J

GraphDB extends the existing RDF and query results formats with dedicated formats that encode embedded triples natively (for example, `<<:subject :predicate :object>>` in Turtle-star). Each new format has its own MIME type and file extension:

RDF-star format	MIME type	File extension
Binary RDF	`application/x-binary-rdf`	brf
Turtle-star	`text/x-turtlestar` `application/x-turtlestar`	ttls
TriG-star	`application/x-trigstar`	trigs
JSON-star query result	`application/x-sparqlstar-results+json`	srjs
TSV-star query result	`text/x-tab-separated-values-star` `application/x-sparqlstar-results+tsv`	tsvs
XML-star query result	`application/x-sparqlstar-results+xml`	xmls

GraphDB uses all RDF-star formats in the way they are defined in RDF4J.

The RDF-star extensions of SPARQL 1.1 Query result formats are in the process of ongoing W3C standardization activities and for this reason may be subject to change. See SPARQL 1.1 Query Results JSON format and SPARQL Query Results XML format for more details.

For the benefit of older clients, in all other formats the embedded triples are serialized as special IRIs in the format `urn:rdf4j:triple:xxx`. Here, `xxx` stands for the Base64 URL-safe encoding of the N-Triples representation of the embedded triple. This is controlled by a boolean writer setting, and is ON by default. The setting is ignored by writers that support RDF-star natively.

Such special IRIs are converted back to triples on parsing. This is controlled by a boolean parser setting, and is ON by default. It is respected by all parsers, including those with native RDF-star support.

7.4.6 Ontologies

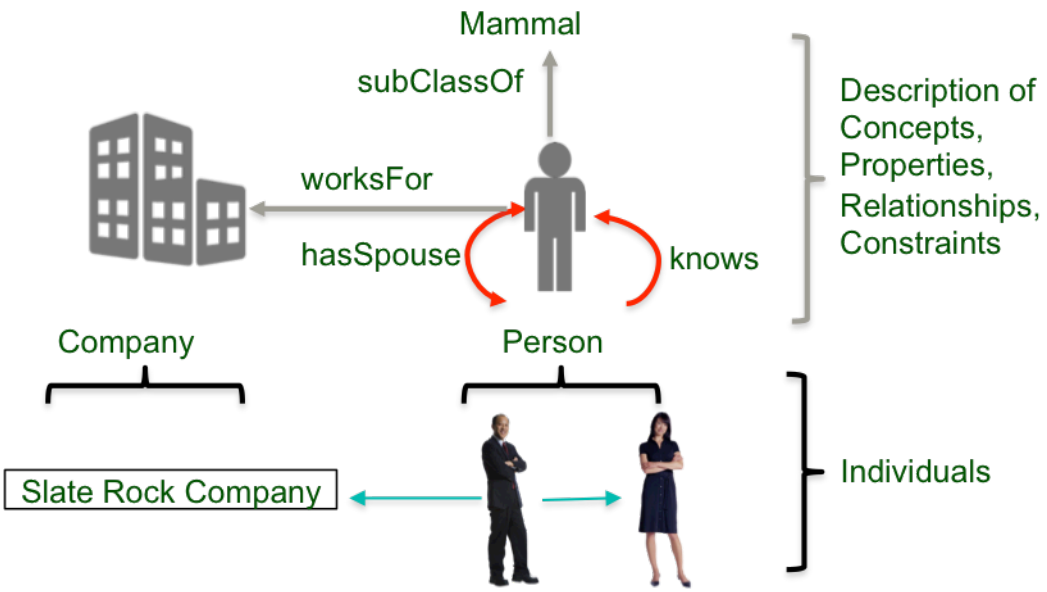
7.4.6.1 What is an ontology?

An ontology is a formal specification that provides shareable and reusable knowledge representation. Examples of ontologies include:

- Taxonomies
- Vocabularies
- Thesauri
- Topic maps
- Logical models

An ontology specification includes descriptions of concepts and properties in a domain, relationships between concepts, constraints on how the relationships can be used and individuals as members of concepts.

In the example below, we can classify the two individuals, Fred and Wilma, in a class of type `Person`, and we also know that a `Person` is a `Mammal`. Fred works for the Slate Rock Company and the Slate Rock Company is of type `Company`, so we also know that `Person worksFor Company`.



7.4.6.2 What are the benefits of developing and using an ontology?

First, ontologies are very useful in gaining a common understanding of information and making assumptions explicit in ways that can be used to support a number of activities.

These provisions, a common understanding of information and explicit domain assumptions, are valuable because ontologies support data integration for analytics, apply domain knowledge to data, support application interoperability, enable model driven applications, reduce time and cost of application development, and improve data quality by improving metadata and provenance.

The Web Ontology Language, or OWL, adds more powerful ontology modeling means to RDF and RDFS. Thus, when used with OWL reasoners, like in GraphDB, it provides consistency checks, such as *are there any logical inconsistencies?* It also provides satisfiability checks, such as *are there classes that cannot have instances?* And OWL provides classification such as *the type of an instance*.

OWL also adds identity equivalence and identity difference, such as *sameAs*, *differentFrom*, *equivalentClass*, and *equivalentProperty*.

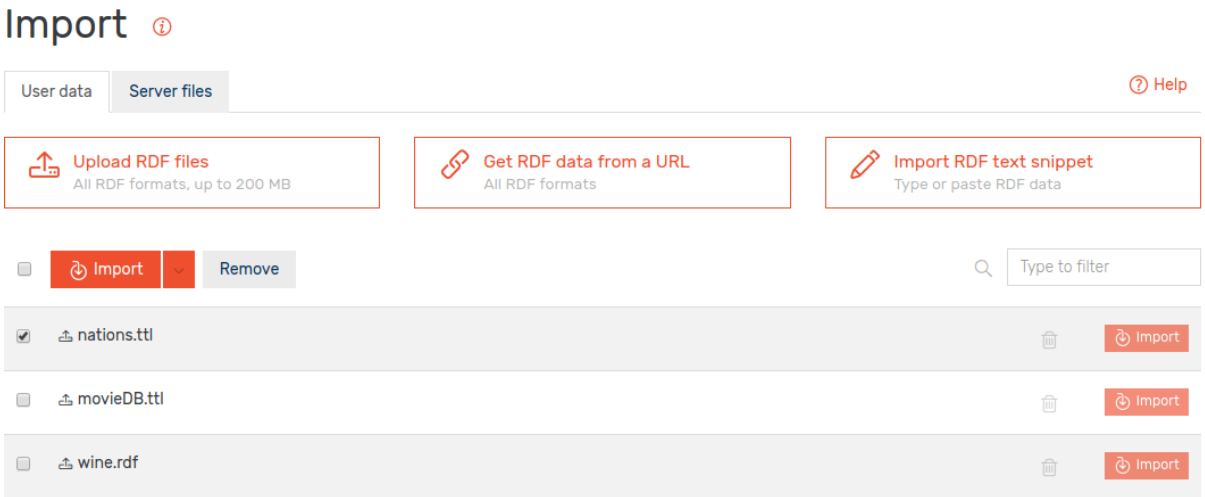
In addition, OWL offers more expressive class definitions, such as class intersection, union, complement, disjointness, and cardinality restrictions.

OWL also offers more expressive property definitions, such as object and datatype properties, transitive, functional, symmetric, inverse properties, and value restrictions.

Finally, ontologies are important because semantic repositories use them as semantic schemata. This makes automated reasoning about the data possible (and easy to implement) since the most essential relationships between the concepts are built into the ontology.

7.4.6.3 Using ontologies in GraphDB

To load your ontology in GraphDB, simply use the import function in the GraphDB Workbench. The example below shows loading an ontology through the *Import* view:



7.4.7 Ontology mapping with the `owl:sameAs` property

GraphDB's `owl:sameAs` optimization is used for mapping the same concepts from two or more datasets, where each of these concepts can have different features and relations to other concepts. In this way, making a union between such datasets provides more complete data. In RDF, concepts are represented with a unique resource name by using a namespace, which is different for every dataset. Therefore, it is more useful to unify all names of a single concept, so that when querying data, you are able to work with concepts rather than names (i.e., IRIs).

For example, when merging four different datasets, you can use the following query on DBpedia to select everything about Sofia:

```
SELECT * {
  {
    <http://dbpedia.org/resource/Sofia> ?p ?o .
  }
  UNION
  {
    <http://data.nytimes.com/nytimes:N82091399958465550531> ?p ?o .
  }
  UNION
  {
    <http://sws.geonames.org/727011/> ?p ?o .
  }
  UNION
  {
```

(continues on next page)

(continued from previous page)

```
    <http://rdf.freebase.com/ns/m/0ftjx> ?p ?o .
  }
}
```

Or you can even use a shorter one:

```
SELECT * {
  ?s ?p ?o
  FILTER (?s IN (
    <http://dbpedia.org/resource/Sofia>,
    <http://data.nytimes.com/nytimes:N82091399958465550531>,
    <http://sws.geonames.org/727011/>,
    <http://rdf.freebase.com/ns/m/0ftjx>))
}
```

As you can see, here Sofia appears with four different URIs, although they denote the same concept. Of course, this is a very simple query. Sofia has also relations to other entities in these datasets, such as Plovdiv, i.e., `<[http://dbpedia.org/resource/Plovdiv]>`, `<[http://sws.geonames.org/653987/]>`, `<[http://rdf.freebase.com/ns/m/1aihge]>`.

What's more, not only the different instances of one concept have multiple names but their properties also appear with many names. Some of them are specific for a given dataset (e.g., GeoNames has longitude and latitude, while DBpedia provides wikilinks) but there are class hierarchies, labels, and other common properties used by most of the datasets.

This means that even for the simplest query, you may have to write the following:

```
SELECT * {
  ?s ?p1 ?x .
  ?x ?p2 ?o .
  FILTER (?s IN (
    <http://dbpedia.org/resource/Sofia>,
    <http://data.nytimes.com/nytimes:N82091399958465550531>,
    <http://sws.geonames.org/727011/>,
    <http://rdf.freebase.com/ns/m/0ftjx>))
  FILTER (?p1 IN (
    <http://dbpedia.org/property/wikilink>,
    <http://sws.geonames.org/p/relatesTo>))
  FILTER (?p2 IN (
    <http://dbpedia.org/property/wikilink>,
    <http://sws.geonames.org/p/relatesTo>))
  FILTER (?o IN (<http://dbpedia.org/resource/Plovdiv>,
    <http://sws.geonames.org/653987/>,
    <http://rdf.freebase.com/ns/m/1aihge>))
}
```

But if you can say through rules and assertions that given URIs are the same, then you can simply write:

```
SELECT * {
  <http://dbpedia.org/resource/Sofia> <http://sws.geonames.org/p/relatesTo> ?x .
  ?x <http://sws.geonames.org/p/relatesTo> <http://dbpedia.org/resource/Plovdiv> .
}
```

If you link two nodes with `owl:sameAs`, the statements that appear with the first node's subject, predicate, and object will be copied, replacing respectively the subject, predicate, and the object that appear with the second node.

For example, given that `<[http://dbpedia.org/resource/Sofia]>` `owl:sameAs` `<[http://data.nytimes.com/N82091399958465550531]>` and also that:

```
<http://dbpedia.org/resource/Sofia> a <http://dbpedia.org/resource/Populated_place> .
<http://data.nytimes.com/N82091399958465550531> a <http://www.opengis.net/gml/_Feature> .
<http://dbpedia.org/resource/Plovdiv> <http://dbpedia.org/property/wikilink> <http://dbpedia.org/
↪resource/Sofia> .
```

then you can conclude with the given rules that:

```
<http://dbpedia.org/resource/Sofia> a <http://www.opengis.net/gml/_Feature> .
<http://data.nytimes.com/N82091399958465550531> a <http://dbpedia.org/resource/Populated_place> .
<http://dbpedia.org/resource/Plovdiv> <http://dbpedia.org/property/wikilink> <http://data.nytimes.com/
↪N82091399958465550531> .
```

The challenge with `owl:sameAs` is that when there are many 'mappings' of nodes between datasets, and especially when big chains of `owl:sameAs` appear, it becomes inefficient. `owl:sameAs` is defined as Symmetric and Transitive, so given that A sameAs B sameAs C, it also follows that A sameAs A, A sameAs C, B sameAs A, B sameAs B, C sameAs A, C sameAs B, C sameAs C. If you have such a chain with N nodes, then N^2 `owl:sameAs` statements will be produced (including the explicit N-1 `owl:sameAs` statements that produce the chain). Also, the `owl:sameAs` rules will copy the statements with these nodes N times, given that each statement contains only one node from the chain and the other nodes are not sameAs anything. But you can also have a statement <S P O> where S sameAs Sx, P sameAs Py, O sameAs Oz, where the `owl:sameAs` statements for S are K, for P are L and for O are M, yielding K*L*M statement copies overall.

Therefore, instead of using these simple rules and axioms for `owl:sameAs` (actually two axioms that state that it is Symmetric and Transitive), GraphDB offers an effective *non-rule implementation*, i.e., the `owl:sameAs` support is hard-coded. The given rules are commented out in the `.pie` files and are left only as a reference.

7.5 Customization and other advanced development

7.5.1 Programming with GraphDB

Note: JavaScript developers may also be interested in graphdb.js , a GraphDB and RDF4J data access library written in JavaScript to be used in Node.js.

GraphDB is built on top of RDF4J, a powerful Java framework for processing and handling RDF data. This includes creating, parsing, storing, inferencing,and querying over such data. It offers an easy-to-use API. GraphDB comes with a set of example programs and utilities that illustrate the basics of accessing GraphDB through the RDF4J API.

7.5.1.1 Installing Maven dependencies

All GraphDB programming examples are provided as a single Maven project. GraphDB is available from Maven Central (the public Maven repository). You can find the most recent version on MVN Repository.

7.5.1.2 Examples

The two examples below can be found under `examples/developer-getting-started` of the GraphDB distribution.

Hello world in GraphDB

The following program opens a connection to a repository, evaluates a SPARQL query and prints the result. The example uses the `GraphDBHTTPRepository` class, which is an extension of RDF4J's `HTTPRepository` that adds support for GraphDB features such as the GraphDB cluster.

In order to run the example program, you need to build from the appropriate `.pom` file:

```
mvn install
```

Followed by running the resultant `.jar` file:

```
java -jar dev-examples-1.0-SNAPSHOT.jar
```

```
package com.ontotext.graphdb.example.app.hello;

import com.ontotext.graphdb.repository.http.GraphDBHTTPRepository;
import com.ontotext.graphdb.repository.http.GraphDBHTTPRepositoryBuilder;
import org.eclipse.rdf4j.model.Value;
import org.eclipse.rdf4j.query.*;
import org.eclipse.rdf4j.repository.RepositoryConnection;

/**
 * Hello World app for GraphDB
 */
public class HelloWorld {
    public void hello() throws Exception {
        // Connect to a remote repository using the GraphDB client API
        // (ruleset is irrelevant for this example)
        GraphDBHTTPRepository repository = new GraphDBHTTPRepositoryBuilder()
                .withServerUrl("http://localhost:7200")
                .withRepositoryId("myrepo")
                //.withCluster(); // uncomment this line to enable cluster mode
                .build();

        // Alternative access to a remote repository using pure RDF4J
        // HTTPRepository repository = new HTTPRepository("http://localhost:7200/repositories/myrepo");

        // Separate connection to a repository
        RepositoryConnection connection = repository.getConnection();

        try {
            // Preparing a SELECT query for later evaluation
            TupleQuery tupleQuery = connection.prepareTupleQuery(QueryLanguage.SPARQL,
                    "SELECT ?x WHERE {" +
                            "BIND('Hello world!' as ?x)" +
                            "}");

            // Evaluating a prepared query returns an iterator-like object
            // that can be traversed with the methods hasNext() and next()
            TupleQueryResult tupleQueryResult = tupleQuery.evaluate();
            while (tupleQueryResult.hasNext()) {
                // Each result is represented by a BindingSet, which corresponds to a result row
                BindingSet bindingSet = tupleQueryResult.next();

                // Each BindingSet contains one or more Bindings
                for (Binding binding : bindingSet) {
                    // Each Binding contains the variable name and the value for this result row
                    String name = binding.getName();
                    Value value = binding.getValue();

                    System.out.println(name + " = " + value);
```

(continues on next page)

(continued from previous page)

```
                }

                // Bindings can also be accessed explicitly by variable name
                //Binding binding = bindingSet.getBinding("x");
            }

            // Once we are done with a particular result we need to close it
            tupleQueryResult.close();

            // Doing more with the same connection object
            // ...
        } finally {
            // It is best to close the connection in a finally block
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        new HelloWorld().hello();
    }
}
```

Family relations app

This example illustrates loading of ontologies and data from files, querying data through SPARQL `SELECT`, deleting data through the RDF4J API and inserting data through SPARQL `INSERT`.

In order to run the example program, you first need to locate appropriate pom file. In this file, there will be a commented line pointing towards the `FamilyRelationsApp` class. Remove the comment markers from this line, making it active, and comment out the line pointing towards the `HelloWorld` class instead. Then build the app from the `.pom` file:

```
mvn install
```

Followed by running the resultant `.jar` file:

```
java -jar dev-examples-1.0-SNAPSHOT.jar
```

```
package com.ontotext.graphdb.example.app.family;

import com.ontotext.graphdb.example.util.QueryUtil;
import com.ontotext.graphdb.example.util.UpdateUtil;

import com.ontotext.graphdb.repository.http.GraphDBHTTPRepository;
import com.ontotext.graphdb.repository.http.GraphDBHTTPRepositoryBuilder;
import org.eclipse.rdf4j.model.IRI;
import org.eclipse.rdf4j.model.impl.SimpleValueFactory;
import org.eclipse.rdf4j.query.*;
import org.eclipse.rdf4j.query.impl.SimpleBinding;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.repository.RepositoryException;
import org.eclipse.rdf4j.rio.RDFFormat;
import org.eclipse.rdf4j.rio.RDFParseException;

import java.io.IOException;

/**
 * An example that illustrates loading of ontologies, data, querying and modifying data.
 */
```

(continues on next page)

(continued from previous page)

```
public class FamilyRelationsApp {
    private RepositoryConnection connection;

    public FamilyRelationsApp(RepositoryConnection connection) {
        this.connection = connection;
    }

    /**
     * Loads the ontology and the sample data into the repository.
     *
     * @throws RepositoryException
     * @throws IOException
     * @throws RDFParseException
     */
    public void loadData() throws RepositoryException, IOException, RDFParseException {
        System.out.println("# Loading ontology and data");

        // When adding data we need to start a transaction
        connection.begin();

        // Adding the family ontology
        connection.add(FamilyRelationsApp.class.getResourceAsStream("/family-ontology.ttl"), "urn:base",
↪ RDFFormat.TURTLE);

        // Adding some family data
        connection.add(FamilyRelationsApp.class.getResourceAsStream("/family-data.ttl"), "urn:base",
↪RDFFormat.TURTLE);

        // Committing the transaction persists the data
        connection.commit();
    }

    /**
     * Lists family relations for a given person. The output will be printed to stdout.
     *
     * @param person a person (the local part of a URI)
     * @throws RepositoryException
     * @throws MalformedQueryException
     * @throws QueryEvaluationException
     */
    public void listRelationsForPerson(String person) throws RepositoryException,
↪MalformedQueryException, QueryEvaluationException {
        System.out.println("# Listing family relations for " + person);

        // A simple query that will return the family relations for the provided person parameter
        TupleQueryResult result = QueryUtil.evaluateSelectQuery(connection,
                "PREFIX family: <http://examples.ontotext.com/family#>" +
                        "SELECT ?p1 ?r ?p2 WHERE {" +
                        "?p1 ?r ?p2 ." +
                        "?r rdfs:subPropertyOf family:hasRelative ." +
                        "FILTER(?r != family:hasRelative)" +
                        "}",
                new SimpleBinding("p1", uriForPerson(person))
        );

        while (result.hasNext()) {
            BindingSet bindingSet = result.next();
            IRI p1 = (IRI) bindingSet.getBinding("p1").getValue();
            IRI r = (IRI) bindingSet.getBinding("r").getValue();
            IRI p2 = (IRI) bindingSet.getBinding("p2").getValue();
```

(continues on next page)

(continued from previous page)

```
            System.out.println(p1.getLocalName() + " " + r.getLocalName() + " " + p2.getLocalName());
        }
        // Once we are done with a particular result we need to close it
        result.close();
    }

    /**
     * Deletes all triples that refer to a person (i.e. where the person is the subject or the object).
     *
     * @param person the local part of a URI referring to a person
     * @throws RepositoryException
     */
    public void deletePerson(String person) throws RepositoryException {
        System.out.println("# Deleting " + person);

        // When removing data we need to start a transaction
        connection.begin();

        // Removing a person means deleting all triples where the person is the subject or the object.
        // Alternatively, this can be done with SPARQL.
        connection.remove(uriForPerson(person), null, null);
        connection.remove((IRI) null, null, uriForPerson(person));

        // Committing the transaction persists the changes
        connection.commit();
    }

    /**
     * Adds a child relation to a person, i.e. inserts the triple :person :hasChild :child.
     *
     * @param child  the local part of a URI referring to a person (the child)
     * @param person the local part of a URI referring to a person
     * @throws MalformedQueryException
     * @throws RepositoryException
     * @throws UpdateExecutionException
     */
    public void addChildToPerson(String child, String person) throws MalformedQueryException,
↪RepositoryException, UpdateExecutionException {
        System.out.println("# Adding " + child + " as a child to " + person);

        IRI childURI = uriForPerson(child);
        IRI personURI = uriForPerson(person);

        // When adding data we need to start a transaction
        connection.begin();

        // We interpolate the URIs inside the string as INSERT DATA may not contain variables (bindings)
        UpdateUtil.executeUpdate(connection,
                String.format(
                        "PREFIX family: <http://examples.ontotext.com/family#>" +
                                "INSERT DATA {" +
                                "<%s> family:hasChild <%s>" +
                                "}", personURI, childURI));

        // Committing the transaction persists the changes
        connection.commit();
    }

    private IRI uriForPerson(String person) {
        return SimpleValueFactory.getInstance().createIRI("http://examples.ontotext.com/family/data#" +
↪person);
```

(continues on next page)

(continued from previous page)

```
    }

    public static void main(String[] args) throws Exception {
        // Connect to a remote repository using the GraphDB client API
        // Note that in order to infer grandparents/grandchildren the repository requires the OWL2-RL
↪ruleset
        GraphDBHTTPRepository repository = new GraphDBHTTPRepositoryBuilder()
                .withServerUrl("http://localhost:7200")
                .withRepositoryId("myrepo")
                //.withCluster(); // uncomment this line to enable cluster mode
                .build();
        // Alternative access to a remote repository using pure RDF4J
        // HTTPRepository repository = new HTTPRepository("http://localhost:7200/repositories/myrepo");

        // Separate connection to a repository
        RepositoryConnection connection = repository.getConnection();

        // Clear the repository before we start
        connection.clear();

        FamilyRelationsApp familyRelations = new FamilyRelationsApp(connection);

        try {
            familyRelations.loadData();

            // Once we've loaded the data we should see all explicit and implicit relations for John
            familyRelations.listRelationsForPerson("John");

            // Let's delete Mary
            familyRelations.deletePerson("Mary");

            // Deleting Mary also removes Kate from John's list of relatives as Kate is his relative
↪through Mary
            familyRelations.listRelationsForPerson("John");

            // Let's add some children to Charles
            familyRelations.addChildToPerson("Bob", "Charles");
            familyRelations.addChildToPerson("Annie", "Charles");

            // After adding two children to Charles John's family is big again
            familyRelations.listRelationsForPerson("John");

            // Finally, let's see Annie's family too
            familyRelations.listRelationsForPerson("Annie");
        } finally {
            // It is best to close the connection in a finally block
            connection.close();
        }
    }
}
```

We also recommend the online book Programming with RDF4J provided by the RDF4J project. It provides detailed explanations on the RDF4J API and its core concepts.

7.5.2 Automated update notifications

7.5.2.1 What are GraphDB local notifications?

Notifications are a publish/subscribe mechanism for registering and receiving events from a GraphDB repository whenever triples matching a certain graph pattern are inserted or removed.

The *RDF4J API* provides such a mechanism where a `RepositoryConnectionListener` can be notified of changes to a `NotifiyingRepositoryConnection`. However, the GraphDB notifications API works at a lower level and uses the internal raw entity IDs for the subject, predicate, and object instead of Java objects. This enables much higher performance. The downside is that the client must do a separate lookup to get the actual entity values. Because of this, the notification mechanism works only when the client is running inside the same JVM as the repository instance.

Note: Local notifications only work in an embedded GraphDB instance, which is usually used only in test environments.

For remote notifications, we recommend using the *Kafka GraphDB Connector*.

7.5.2.2 How to register for local notifications

To receive notifications, register by providing a SPARQL query using a `SPARQLQueryListener` as demonstrated below.

Note: The SPARQL query is interpreted as a plain graph pattern by ignoring more complicated SPARQL constructs such as `FILTER`, `OPTIONAL`, `DISTINCT`, `LIMIT`, and `ORDER BY`. Therefore, the SPARQL query is interpreted as a complex graph pattern involving triple patterns combined by means of joins and unions at any level. The order of the triple patterns is not significant.

Here is an example of how to register for notifications based on a given SPARQL query:

```
AbstractRepository rep =
    ((OwlimSchemaRepository)owlimSail).getRepository();
EntityPool ent = ((OwlimSchemaRepository)owlimSail).getEntities();
String query = "SELECT * WHERE { ?s rdf:type ?o }";
SPARQLQueryListener listener =
    new SPARQLQueryListener(query, rep, ent) {
        public void notifyMatch(int subj, int pred, int obj, int context) {
            System.out.println("Notification on subject: " + subj);
        }
    }
rep.addListener(listener);    // start receiving notifications
...
rep.removeListener(listener); // stop receiving notifications
```

In the example code, the caller will be asynchronously notified about incoming statements that match the pattern `?s rdf:type ?o`.

Note: In general, notifications are sent for all incoming triples that contribute to a solution of the query. You can map the integer parameters in the `notifyMatch` method to values using the `EntityPool` object. Any statements inferred from newly inserted statements are also subject to handling by the notification mechanism—that is, clients are also notified of new implicit statements when the requested triple pattern matches.

Note: The subscriber should not rely on any particular order or distinctness of the statement notifications. Duplicate statements might be delivered in response to a graph pattern subscription in an order that is not even bound to

the chronological order of the insertion of the statements in the underlying triplestore.

Tip: The purpose of the notification services is to enable the efficient and timely discovery of newly added RDF data. Therefore, it should be treated as a mechanism for giving the client a hint that certain new data is available and not as an asynchronous SPARQL evaluation engine.

7.5.3 Extending GraphDB Workbench

GraphDB Workbench is now a separate open-source project, enabling the fast development of knowledge graph prototypes or rich UI applications. This provides you with the ability to add your custom colors to the graph views, as well as to easily start a FactForge-like interface.

This tutorial will show you how to extend and customize GraphDB Workbench by adding your own page and Angular controller. We will create a simple paths application that allows you to import RDF data, find paths between to nodes in the graph, and visualize them using D3.

7.5.3.1 Clone, download, and run GraphDB Workbench

1. Download and run GraphDB 9.x on the default port 7200.
2. Clone the GraphDB Workbench project from GitHub.
3. Enter the project directory and execute `npm install` in order to install all necessary dependencies locally.
4. Run `npm run start` to start a webpack development server that proxies REST requests to `localhost:7200`:

```
git clone https://github.com/Ontotext-AD/graphdb-workbench.git graphdb-workbench-paths
cd graphdb-workbench-paths
git checkout <branch>
npm install
npm run start
```

Now GraphDB Workbench is opened on `http://localhost:9000/`.

7.5.3.2 Add your own page and controller

All pages are located under `src/pages/`, so you need to add your new page `paths.html` there with a *`{title}`* placeholder. The page content will be served by an Angular controller, which is placed under `src/js/angular/graphexplore/controllers/paths.controller.js`. Path exploration is a functionality related to graph exploration, so you need to register your new page and controller there.

In `src/js/angular/graphexplore/app.js`:

1. Import the controller:

```
'angular/graphexplore/controllers/paths.controller',
```

2. Add it to the route provider:

```
.when('/paths', {
        templateUrl: 'pages/paths.html',
        controller: 'GraphPathsCtrl',
        title: 'Graph Paths',
        helpInfo: 'Find all paths in a graph.',
});
```

3. And register it in the menu:

```
$menuItemsProvider.addItem({
    label: 'Paths',
    href: 'paths',
    order: 5,
    parent: 'Explore',
});
```

Now you can see your page in GraphDB Workbench.

Next, let's create the paths controller itself.

1. In the `paths.controller.js` that you created, add:

```
define([
        'angular/core/services',
        'lib/common/d3-utils'
    ],
    function (require, D3) {
        angular
            .module('graphdb.framework.graphexplore.controllers.paths', [
                'toastr',
                'ui.bootstrap',
            ])
            .controller('GraphPathsCtrl', GraphPathsCtrl);

        GraphPathsCtrl.$inject = ["$scope", "$rootScope", "$repositories", "toastr", "
↪$timeout", "$http", "ClassInstanceDetailsService", "AutocompleteService", "$q", "
↪$location"];

        function GraphPathsCtrl($scope, $rootScope, $repositories, toastr, $timeout,
↪$http, ClassInstanceDetailsService, AutocompleteService, $q, $location) {

        }
    }
);
```

2. And register the module in `src/js/angular/graphexplore/modules.js`

```
'graphdb.framework.graphexplore.controllers.paths',
```

Now your controller and page are ready to be filled with content.

7.5.3.3 Add repository checks

In your page, you need a repository with data in it. Like most views in GraphDB, you need to have a repository set. The template that most of the pages use is similar to this, where the `repository-is-set` div is where you put your html. Error handling related to repository errors is added for you.

```
<div class="container-fluid">
    <h1>
        {{title}}
        <span class="btn btn-link"
              popover-template="'js/angular/templates/titlePopoverTemplate.html'"
              popover-trigger="mouseenter"
              popover-placement="bottom-right"
              popover-append-to-body="true"><span class="icon-info"></span></span>
    </h1>
    <div core-errors></div>
    <div system-repo-warning></div>
    <div class="alert alert-danger" ng-show="repositoryError">
        <p>The currently selected repository cannot be used for queries due to an error:</p>
```

(continues on next page)

(continued from previous page)

```
        <p>{{repositoryError}}</p>
    </div>

    <div id="repository-is-set" ng-show="getActiveRepository() && !isLoadingLocation() &&
↪hasActiveLocation() && 'SYSTEM' !== getActiveRepository()">
        {{getActiveRepository()}}
    </div>
</div>
```

You need to define the functions on which this snippet depends in your `paths.controller.js`. They use the repository service that you imported in the controller definition.

```
$scope.getActiveRepository = function () {
    return $repositories.getActiveRepository();
};

$scope.isLoadingLocation = function () {
    return $repositories.isLoadingLocation();
};

$scope.hasActiveLocation = function () {
    return $repositories.hasActiveLocation();
};
```

7.5.3.4 Repository setup

1. Create a repository.
2. Import the `airports.ttl` dataset.
3. Enable the *Autocomplete index* for your repository.
4. Execute the following SPARQL insert to add direct links for flights:

```
PREFIX onto: <http://www.ontotext.com/>
INSERT {
  ?node onto:hasFlightTo ?destination .
}  WHERE {
    ?flight <http://openflights.org/resource/route/sourceId> ?node .
    ?flight <http://openflights.org/resource/route/destinationId> ?destination .
}
```

Now we will search for paths between airports based on the `hasFlightTo` predicate.

7.5.3.5 Select departure and destination airport

Now let's add inputs using Autocomplete to select the departure and destination airports. Inside the `repository-is-set` diff, add the two fields. Note the `visual-callback="findPath(startNode, uri)"` snippet that defines the callback to be executed once a value is selected through the Autocomplete. `uri` is the value from the Autocomplete. The following code sets the `starNode` variable in Angular and calls the `findPath` function when the destination is given. You can find out how to define this function in the scope a little further down in this tutorial.

```
<div class="card mb-2">
    <div class="card-block">
        <h3>From</h3>
        <p>Search for a start node</p>
        <search-resource-input class="search-rdf-resources"
                               namespacespromise="getNamespacesPromise"
                               autocompletepromisestatus="getAutocompletePromise"
```

(continues on next page)

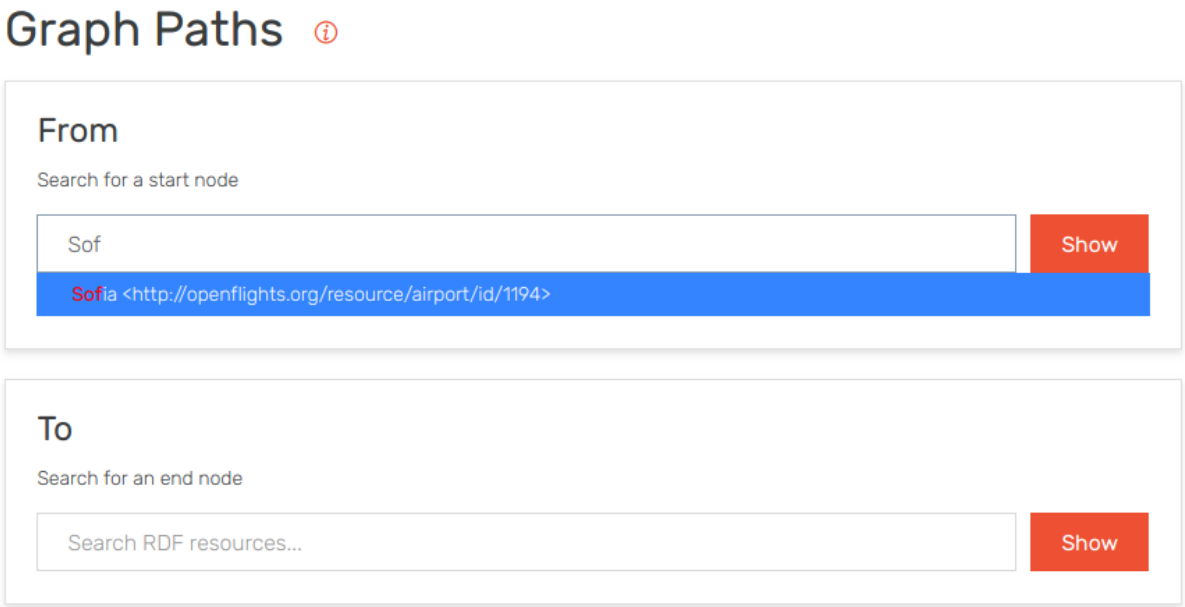
(continued from previous page)

```
                                text-button=""
                                visual-button="Show"
                                visual-callback="startNode = uri"
                                empty="empty"
                                preserve-input="true">
        </search-resource-input>
    </div>
</div>
<div class="card mb-2">
    <div class="card-block">
        <h3>To</h3>
        <p>Search for an end node</p>
        <search-resource-input class="search-rdf-resources"
                                namespacespromise="getNamespacesPromise"
                                autocompletepromisestatus="getAutocompletePromise"
                                text-button=""
                                visual-button="Show"
                                visual-callback="findPath(startNode, uri)"
                                empty="empty"
                                preserve-input="true">
        </search-resource-input>
    </div>
</div>
```

They need the `getNamespacesPromise` and `getAutocompletePromise` to fetch the Autocomplete data. They should be initialized once the repository has been set in the controller.

```
function initForRepository() {
    if (!$repositories.getActiveRepository()) {
        return;
    }
    $scope.getNamespacesPromise = ClassInstanceDetailsService.getNamespaces($scope.
↪getActiveRepository());
    $scope.getAutocompletePromise = AutocompleteService.checkAutocompleteStatus();
}

$scope.$on('repositoryIsSet', function(event, args) {
    initForRepository();
});
initForRepository();
```

Note that both of these functions need to be called when the repository is changed, because you need to make sure that Autocomplete is enabled for this repository, and fetch the namespaces for it. Now you can autocomplete in your page.



7.5.3.6 Find the paths between the selected airports

Now let's implement the `findPath` function in the scope. It finds all paths between nodes by using a simple depth-first search algorithm (recursive algorithm based on the idea of backtracking).

For each node, you can obtain its siblings with a call to the `rest/explore-graph/links` endpoint. This is the same endpoint the Visual graph is using to expand node links. Note that it is not part of the GraphDB API, but we will reuse it for simplicity.

As an alternative, you can also obtain the direct links of a node by sending a SPARQL query to GraphDB.

Note: This is a demo implementation. For each repository containing a lot of links, the proposed approach is not appropriate, as it will send a request to the server for each node. This will quickly result in a huge amount of requests, which will very soon flood the browser.

```
var maxPathLength = 3;

var findPath = function (startNode, endNode, visited, path) {
    // A path is found, return a promise that resolves to it
    if (startNode === endNode) {
        return $q.when(path)
    }
    // Find only paths with maxLength, we want to cut only short paths between airports
    if (path.length === maxPathLength) {
        return $q.when([])
    }
    return $http({
        url: 'rest/explore-graph/links',
        method: 'GET',
        params: {
            iri: startNode,
            config: 'default',
            linksLimit: 50
        }
    }).then(function (response) {
        // Use only links with the hasFlightTo predicate
        var flights = _.filter(response.data, function(r) {return r.predicates[0] == "hasFlightTo"});
        // For each links, continue to search path recursively
```

(continues on next page)

(continued from previous page)

```
        var promises = _.map(flights, function (link) {
            var o = link.target;
            if (!visited.includes(o)) {
                return findPath(o, endNode, visited.concat(o), path.concat(link));
            }
            return $q.when([]);
        });
        // Group together all promises that resolve to paths
        return $q.all(promises);
    }, function (response) {
        var msg = getError(response.data);
        toastr.error(msg, 'Error looking for path node');
    });
}

$scope.findPath = function (startNode, endNode) {
    findPath(startNode, endNode, [startNode], []).then(function (linksFound) {
        renderGraph(_.flattenDeep(linksFound));
    });
}
```

The `findPath` recursive function returns all the promises that will or will not resolve to paths. Each path is a collection of links.

When all promises are resolved, you can flatten the array to obtain all links from all paths and draw one single graph with these links. Graph drawing is done with D3 in the `renderGraph` function. It needs a `graph-visualization` element to draw the graph inside. Add it inside the `repository-is-set` element below the autocomplete divs.

Additionally, import `graphs-visualizations.css` to reuse some styles.

```
<div class="card mb-2">
    ..
</div>
<div class="card mb-2">
    ...
</div>
<div class="graph-visualization"></div>
```

```
<link href="css/graphs-vizualizations.css" rel="stylesheet"/>
```

Now add the `renderGraph` render function mentioned above:

```
var width = 1000,
    height = 1000;

var nodeLabelRectScaleX = 1.75;

var force = d3.layout.force()
    .gravity(0.07)
    .size([width, height]);

var svg = d3.select(".main-container .graph-visualization").append("svg")
    .attr("viewBox", "0 0 " + width + " " + height)
    .attr("preserveAspectRatio", "xMidYMid meet");

function renderGraph(linksFound) {
    var graph = new Graph();

    var nodesFromLinks = _.union(_.flatten(_.map(linksFound, function (d) {
        return [d.source, d.target];
    })));
```

(continues on next page)

(continued from previous page)

```
    var promises = [];
    var nodesData = [];

    // For each node in the graph find its label with a rest call
    _.forEach(nodesFromLinks, function (newNode, index) {
        promises.push($http({
            url: 'rest/explore-graph/node',
            method: 'GET',
            params: {
                iri: newNode,
                config: 'default',
                includeInferred: true,
                sameAsState: true
            }
        }).then(function (response) {
            // Save the data for later
            nodesData[index] = response.data;
        }));
    });

    // Waits for all of the collected promises and then:
    // - adds each new node
    // - redraws the graph
    $q.all(promises).then(function () {
        _.forEach(nodesData, function (nodeData, index) {
            // Calculate initial positions for the new nodes based on spreading them evenly
            // on a circle.
            var theta = 2 * Math.PI * index / nodesData.length;
            var x = Math.cos(theta) * height / 3;
            var y = Math.sin(theta) * height / 3;
            graph.addNode(nodeData, x, y);
        });

        graph.addLinks(linksFound);
        draw(graph);
    });
}

function Graph() {
    this.nodes = [];
    this.links = [];

    this.addNode = function (node, x, y) {
        node.x = x;
        node.y = y;
        this.nodes.push(node);
        return node;
    };

    this.addLinks = function (newLinks) {
        var nodes = this.nodes;
        var linksWithNodes = _.map(newLinks, function (link) {
            return {
                "source": _.find(nodes, function (o) {
                    return o.iri === link.source;
                }),
                "target": _.find(nodes, function (o) {
                    return o.iri === link.target;
                }),
                "predicates": link.predicates
```

(continues on next page)

(continued from previous page)

```
            };
        });
        Array.prototype.push.apply(this.links, linksWithNodes);
    };
}

// Draw the graph using d3 force layout
function draw(graph) {
    d3.selectAll("svg g").remove();

    var container = svg.append("g").attr("class", "nodes-container");

    var link = svg.selectAll(".link"),
        node = svg.selectAll(".node");

    force.nodes(graph.nodes).charge(-3000);
    force.links(graph.links).linkDistance(function (link) {
        // link distance depends on length of text with an added bonus for strongly connected nodes,
        // i.e. they will be pushed further from each other so that their common nodes can cluster up
        return getPredicateTextLength(link) + 30;
    });

    function getPredicateTextLength(link) {
        var textLength = link.source.size * 2 + link.target.size * 2 + 50;
        return textLength * 0.75;
    }


    // arrow markers
    container.append("defs").selectAll("marker")
        .data(force.links())
        .enter().append("marker")
        .attr("class", "arrow-marker")
        .attr("id", function (d) {
            return d.target.size;
        })
        .attr("viewBox", "0 -5 10 10")
        .attr("refX", function (d) {
            return d.target.size + 11;
        })
        .attr("refY", 0)
        .attr("markerWidth", 10)
        .attr("markerHeight", 10)
        .attr("orient", "auto")
        .append("path")
        .attr("d", "M0,-5L10,0L0,5 L10,0 L0, -5");

    // add the links, nodes, predicates and node labels
    var link = container.selectAll(".link")
        .data(graph.links)
        .enter().append("g")
        .attr("class", "link-wrapper")
        .attr("id", function (d) {
            return d.source.iri + '>' + d.target.iri;
        })
        .append("line")
        .attr("class", "link")
        .style("stroke-width", 1)
        .style("fill", "transparent")
        .style("marker-end", function (d) {
            return "url(" + $location.absUrl() + "#" + d.target.size + ")";
```

(continues on next page)

(continued from previous page)

```
        });

    var predicate = container.selectAll(".link-wrapper")
        .append("text")
        .text(function (d, index) {
            return d.predicates[0];
        })
        .attr("class", function (d) {
            if (d.predicates.length > 1) {
                return "predicates";
            }
            return "predicate";
        })
        .attr("dy", "-0.5em")
        .style("text-anchor", "middle")
        .style("display", "")
        .on("mouseover", function (d) {
            d3.event.stopPropagation();
        });

    var node = container.selectAll(".node")
        .data(graph.nodes)
        .enter().append("g")
        .attr("class", "node-wrapper")
        .attr("id", function (d) {
            return d.iri;
        })
        .append("circle")
        .attr("class", "node")
        .attr("r", function (d) {
            return d.size;
        })
        .style("fill", function (d) {
            return "rgb(255, 128, 128)";
        })

    var nodeLabels = container.selectAll(".node-wrapper").append("foreignObject")
        .style("pointer-events", "none")
        .attr("width", function (d) {
            return d.size * 2 * nodeLabelRectScaleX;
        });
    // height will be computed by updateNodeLabels

    updateNodeLabels(nodeLabels);

    function updateNodeLabels(nodeLabels) {
        nodeLabels.each(function (d) {
            d.fontSize = D3.Text.calcFontSizeRaw(d.labels[0].label, d.size, 16, true);
        })
            .attr("height", function (d) {
                return d.fontSize * 3;
            })
            // if this was kosher we would use xhtml:body here but if we do that angular (or the browser)
            // goes crazy and resizes/messes up other unrelated elements. div seems to work too.
            .append("xhtml:div")
            .attr("class", "node-label-body")
            .style("font-size", function (d) {
                return d.fontSize + 'px';
            })
            .append('xhtml:div')
            .text(function (d) {
```

(continues on next page)

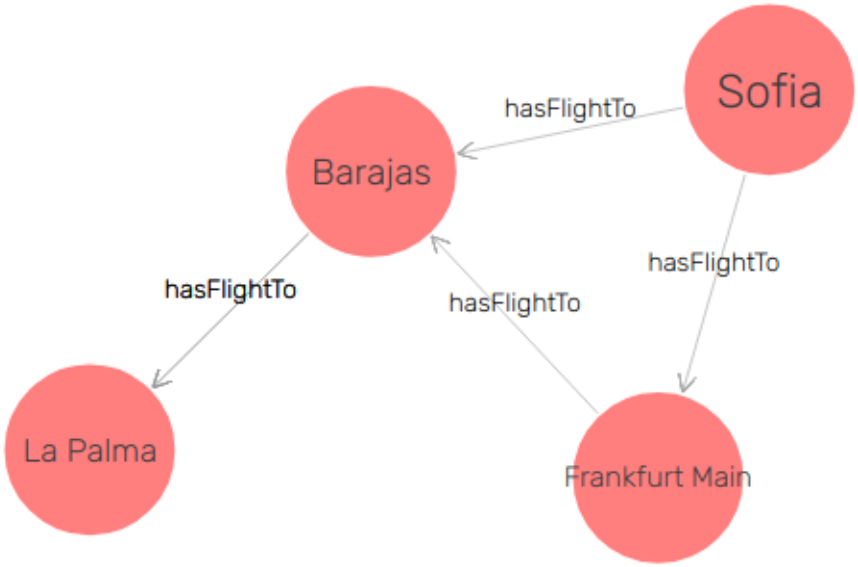
(continued from previous page)

```
                return d.labels[0].label;
            });
    }


    // Update positions on tick
    force.on("tick", function () {

        // recalculate links attributes
        link.attr("x1", function (d) {
            return d.source.x;
        }).attr("y1", function (d) {
            return d.source.y;
        }).attr("x2", function (d) {
            return d.target.x;
        }).attr("y2", function (d) {
            return d.target.y;
        });

        // recalculate predicates attributes
        predicate.attr("x", function (d) {
            return d.x = (d.source.x + d.target.x) * 0.5;
        }).attr("y", function (d) {
            return d.y = (d.source.y + d.target.y) * 0.5;
        });

        // recalculate nodes attributes
        node.attr("cx", function (d) {
            return d.x;
        }).attr("cy", function (d) {
            return d.y;
        });


        nodeLabels.attr("x", function (d) {
            return d.x - (d.size * nodeLabelRectScaleX);
        }).attr("y", function (d) {
            // the height of the nodeLabel box is 3 times the fontSize computed by updateNodeLabels
            // and we want to offset it so that its middle matches the centre of the circle, hence
↪divided by 2
            return d.y - 3 * d.fontSize / 2;
        });

    });
    force.start();
}
```

It obtains the URIs for the nodes from all links, and finds their labels through calls to the `rest/explore-graph/node` endpoint. A graph object is defined to represent the visual abstraction, which is simply a collection of nodes and links. The `draw(graph)` function does the D3 drawing itself using the D3 force layout.

7.5.3.7 Visualize results

Now let's find all paths between Sofia and La Palma with maximum 2 nodes in between (maximum path length 3):



Note: The airports graph is highly connected. Increasing the maximum path length will send too many requests to the server. The purpose of this tutorial is to introduce you to the Workbench extension with a naive paths prototype.

7.5.3.8 Add status message

Noticing that path finding can take some time, we may want to add a message for the user.

```
$scope.findPath = function (startNode, endNode) {
    $scope.pathFinding = true;
    findPath(startNode, endNode, [startNode], []).then(function (linksFound) {
        $scope.pathFinding = false;
        renderGraph(_.flattenDeep(linksFound));
    });
}
```

```
<div ng-show="pathFinding">Looking for all paths between nodes...</div>
<div class="graph-visualization"></div>
```

The source code for this example can be found in the workbench-paths-example GitHub project.

7.5.4 Tuning query behavior

7.5.4.1 What are named graphs

Tip: GraphDB supports the following SPARQL specifications:

- SPARQL 1.1 Protocol for RDF
- SPARQL 1.1 Query
- SPARQL 1.1 Update
- SPARQL 1.1 Federation
- SPARQL 1.1 Graph Store HTTP Protocol

An RDF database can store collections of RDF statements (triples) in separate graphs identified (named) by a URI. A group of statements with a unique name is called a 'named graph'. An RDF database has one more graph, which does not have a name, and it is called the 'default graph'.

The SPARQL query syntax provides a means to execute queries across default and named graphs using FROM and FROM NAMED clauses. These clauses are used to build an RDF dataset, which identifies what statements the SPARQL query processor will use to answer a query. The dataset contains a default graph and named graphs and is constructed as follows:

- `FROM <uri>` — brings statements from the database graph, identified by URI, to the dataset's default graph, i.e., the statements 'lose' their graph name.
- `FROM NAMED <uri>` — brings the statements from the database graph, identified by URI, to the dataset, i.e., the statements keep their graph name.

If either `FROM` or `FROM NAMED` are used, the database's default graph is no longer used as input for processing this query. In effect, the combination of FROM and FROM NAMED clauses exactly defines the dataset. This is somewhat bothersome, as it precludes the possibility, for instance, of executing a query over just one named graph and the default graph. However, there is a programmatic way to get around this limitation as described below.

The default SPARQL dataset

Note: The SPARQL specification does not define what happens when no `FROM` or `FROM NAMED` clauses are present in a query, i.e., it does not define how a SPARQL processor should behave when no dataset is defined. In this situation, implementations are free to construct the default dataset as necessary.

GraphDB constructs the default dataset as follows:

- The dataset's default graph contains the merge of the database's default graph AND all the database named graphs;
- The dataset contains all named graphs from the database.

This means that if a statement `ex:x ex:y ex:z` exists in the database in the graph `ex:g`, then the following query patterns will behave as follows:

Query	Bindings
`SELECT * { ?s ?p ?o }`	`?s=ex:x ?p=ex:y ?o=ex:z`
`SELECT * { GRAPH ?g { ?s ?p ?o } }`	`?s=ex:x ?p=ex:y ?o=ex:z ?g=ex:g`

In other words, the triple `ex:x ex:y ex:z` will appear to be in both the default graph and the named graph `ex:g`.

There are two reasons for this behavior:

1. It provides an easy way to execute a triple pattern query over all stored RDF statements.
2. It allows all named graph names to be discovered, i.e., with this query: `SELECT ?g { GRAPH ?g { ?s ?p ?o } }`.

7.5.4.2 How to manage explicit and implicit statements

GraphDB maintains two flags for each statement:

- **Explicit**: the statement is inserted in the database by the user, using SPARQL UPDATE, the RDF4J API or the *imports configuration parameter* configuration parameter. The same explicit statement can exist in the database's default graph and in each named graph.
- **Implicit**: the statement is created as a result of inference, by either *Axioms* or *Rules*. Inferred statements are ALWAYS created in the database's default graph.

These two flags are not mutually exclusive. The following sequences of operations are possible:

- For the operations, use the names 'insert/delete' for explicit, and 'infer/retract' for implicit (retract means that all premises of the statement are deleted or retracted).
- To show the results after each operation, use tuples `<statement graph flags>` :
  - `<s G EI>` means statement `s` in graph `G` having both flags Explicit and Implicit;
  - `<s _ EI>` means statement `s` in the default graph having both flags Explicit and Implicit;
  - `<_ G _>` means the statement is deleted from graph `G`.

First, let's consider operations on statement `s` in the default graph only:

- insert `<s _ E>`, infer `<s _ EI>`, delete `<s _ I>`, retract `<_ _ _>`;
- insert `<s _ E>`, infer `<s _ EI>`, retract `<s _ E>`, delete `<_ _ _>`;
- infer `<s _ I>`, insert `<s _ EI>`, delete `<s _ I>`, retract `<_ _ _>`;
- infer `<s _ I>`, insert `<s _ EI>`, retract `<s _ E>`, delete `<_ _ _>`;
- insert `<s _ E>`, insert `<s _ E>`, delete `<_ _ _>`;
- infer `<s _ I>`, infer `<s _ I>`, retract `<_ _ _>` (if the two inferences are from the same premises).

This does not show all possible sequences, but it shows the principles:

- No duplicate statement can exist in the default graph;
- Delete/retract clears the appropriate flag;
- The statement is deleted only after both flags are cleared;
- Deleting an inferred statement has no effect (except to clear the `I` flag, if any);
- Retracting an inserted statement has no effect (except to clear the `E` flag, if any);
- Inserting the same statement twice has no effect: insert is idempotent;
- Inferring the same statement twice has no effect: infer is idempotent, and `I` is a flag, not a counter, but the Retraction algorithm ensures `I` is cleared only after all premises of `s` are retracted.

Now, let's consider operations on statement `s` in the named graph `G`, and inferred statement `s` in the default graph:

- insert `<s G E>`, infer `<s _ I> <s G E>`, delete `<s _ I>`, retract `<_ _ _>`;
- insert `<s G E>`, infer `<s _ I> <s G E>`, retract `<s G E>`, delete `<_ _ _>`;
- infer `<s _ I>`, insert `<s G E> <s _ I>`, delete `<s _ I>`, retract `<_ _ _>`;
- infer `<s _ I>`, insert `<s G E> <s _ I>`, retract `<s G E>`, delete `<_ _ _>`;
- insert `<s G E>`, insert `<s G E>`, delete `<_ _ _>`;
- infer `<s _ I>`, infer `<s _ I>`, retract `<_ _ _>` (if the two inferences are from the same premises).

The additional principles here are:

- The same statement can exist in several graphs — as explicit in graph `G` and implicit in the default graph;
- Delete/retract works on the appropriate graph.

Note: In order to avoid a proliferation of duplicate statements, it is recommended not to insert inferable statements in named graphs.

7.5.4.3 How to query explicit and implicit statements

The database's default graph can contain a mixture of explicit and implicit statements. The RDF4J API provides a flag called 'includeInferred', which is passed to several API methods and when set to `false` causes only explicit statements to be iterated or returned. When this flag is set to `true`, both explicit and implicit statements are iterated or returned.

GraphDB provides extensions for more control over the processing of explicit and implicit statements. These extensions allow the selection of explicit, implicit or both for query answering and also provide a mechanism for identifying which statements are explicit and which are implicit. This is achieved by using some 'pseudo-graph' names in `FROM` and `FROM NAMED` clauses, which cause certain flags to be set.

The details are as follows:

`FROM <http://www.ontotext.com/explicit>`

The dataset's default graph includes only explicit statements from the database's default graph.

`FROM <http://www.ontotext.com/implicit>`

The dataset's default graph includes only inferred statements from the database's default graph.

`FROM NAMED <http://www.ontotext.com/explicit>`

The dataset contains a named graph `http://www.ontotext.com/explicit` that includes only explicit statements from the database's default graph, i.e., quad patterns such as GRAPH ?g {?s ?p ?o} rebind explicit statements from the database's default graph to a graph named `http://www.ontotext.com/explicit`.

`FROM NAMED <http://www.ontotext.com/implicit>`

The dataset contains a named graph `http://www.ontotext.com/implicit` that includes only implicit statements from the database's default graph.

Note: These clauses do not affect the construction of the default dataset in the sense that using any combination of the above will still result in a dataset containing all named graphs from the database. All it changes is which statements appear in the dataset's default graph and whether any extra named graphs (explicit or implicit) appear.

7.5.4.4 How to specify the dataset programmatically

The RDF4J API provides an interface `Dataset` and an implementation class `DatasetImpl` for defining the dataset for a query by providing the URIs of named graphs and adding them to the default graphs and named graphs members. This permits `null` to be used to identify the default database graph (or `null context` to use RDF4J terminology).

```
DatasetImpl dataset = new DatasetImpl();
dataset.addDefaultGraph(null);
dataset.addNamedGraph(valueFactory.createURI("http://example.com/g1"));
```

This dataset can then be passed to queries or updates, e.g.:

```
TupleQuery query = connection.prepareTupleQuery(QueryLanguage.SPARQL, queryString);
query.setDataset(dataset);
```

7.5.4.5 How to access internal identifiers for entities

Internally, GraphDB uses integer identifiers (IDs) to index all entities (URIs, blank nodes, literals, and RDF-star [formerly RDF*] embedded triples). Statement indexes are made up of these IDs and a large data structure is used to map from ID to entity value and back. There are occasions (e.g., when interfacing to an application infrastructure) when having access to these internal IDs can improve the efficiency of data structures external to GraphDB by allowing them to be indexed by an integer value rather than a full URI.

Here, we introduce a special GraphDB predicate and function that provide access to the internal IDs. The datatype of the internal IDs is `<http://www.w3.org/2001/XMLSchema#long>`.

Predicate	`<http://www.ontotext.com/owlim/entity#id>`
Description	A map between an entity and an internal ID

Example	Select all entities and their IDs:
	`PREFIX ent: <http://www.ontotext.com/owlim/entity#>`
	`SELECT * WHERE {`
	`?s ent:id ?id`
	`} ORDER BY ?id`

Function	`<http://www.ontotext.com/owlim/entity#id>`
Description	Return an entity's internal ID

Example	Select all statements and order them by the internal ID of the object values:
	`PREFIX ent: <http://www.ontotext.com/owlim/entity#>`
	`SELECT * WHERE {`
	`?s ?p ?o .`
	`} order by ent:id(?o)`

Examples

- Enumerate all entities and bind the nodes to `?s` and their IDs to `?id`, `order by ?id`:

```
select * where {
  ?s <http://www.ontotext.com/owlim/entity#id> ?id
} order by ?id
```

- Enumerate all non-literals and bind the nodes to `?s` and their IDs to `?id`, `order by ?id`:

```
SELECT * WHERE {
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .
  FILTER (!isLiteral(?s)) .
} ORDER BY ?id
```

- Find the internal IDs of subjects of statements with specific predicate and object values:

```
SELECT * WHERE {
  ?s <http://test.org#Pred1> "A literal".
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .
} ORDER BY ?id
```

- Find all statements where the object has the given internal ID by using an explicit, untyped value as the ID (the `"115"` is used as object in the second statement pattern):

```
SELECT * WHERE {
  ?s ?p ?o.
  ?o <http://www.ontotext.com/owlim/entity#id> "115" .
}
```

- As above, but using an `xsd:long` datatype for the constant within a `FILTER` condition:

```
SELECT * WHERE {
  ?s ?p ?o.
  ?o <http://www.ontotext.com/owlim/entity#id> ?id .
  FILTER (?id="115"^^<http://www.w3.org/2001/XMLSchema#long>) .
} ORDER BY ?o
```

- Find the internal IDs of subject and object entities for all statements:

```
SELECT * WHERE {
  ?s ?p ?o.
  ?s <http://www.ontotext.com/owlim/entity#id> ?ids.
  ?o <http://www.ontotext.com/owlim/entity#id> ?ido.
}
```

- Retrieve all statements where the ID of the subject is equal to `"115"^^xsd:long`, by providing an internal ID value within a filter expression:

```
SELECT * WHERE {
  ?s ?p ?o.
  FILTER ((<http://www.ontotext.com/owlim/entity#id>(?s))
               = "115"^^<http://www.w3.org/2001/XMLSchema#long>).
}
```

- Retrieve all statements where the string-ized ID of the subject is equal to `"115"`, by providing an internal ID value within a filter expression:

```
SELECT * WHERE {
  ?s ?p ?o.
  FILTER (str( <http://www.ontotext.com/owlim/entity#id>(?s) ) = "115").
}
```

7.5.4.6 How to use RDF4J 'direct hierarchy' vocabulary

GraphDB supports the RDF4J specific vocabulary for determining 'direct' subclass, subproperty and type relationships. The special vocabulary used and their definitions are shown below. The three predicates are all defined using the namespace definition:

```
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
```

Predicate	Definition
`A sesame:directSubClassOf B`	Class A is a direct subclass of B if: 1. A is a subclass of B and; 2. A and B are not equal and; 3. there is no class C (not equal to A or B) such that A is a subclass of C and C of B.
`P sesame:directSubPropertyOf Q`	Property P is a direct subproperty of Q if: 1. P is a subproperty of Q and; 2. P and Q are not equal and; 3. there is no property R (not equal to P or Q) such that P is a subproperty of R and R of Q.
`I sesame:directType T`	Resource I is a direct type of T if: 1. I is of type T and 2. There is no class U (not equal to T) such that: a. U is a subclass of T and; b. I is of type U.

7.5.4.7 Other query behavior customizations

There are several more special graph URIs in GraphDB, which are used for controlling query evaluation.

`FROM` / `FROM NAMED <http://www.ontotext.com/disable-sameAs>`

Switch off the enumeration of equivalence classes produced by the *Optimization of owl:sameAs*. By default, all `owl:sameAs` URIs are returned by triple pattern matching. This clause reduces the number of results to include a single representative from each `owl:sameAs` class. For more details, see *Not enumerating sameAs*.

`FROM` / `FROM NAMED <http://www.ontotext.com/count>`

Used for triggering the evaluation of the query, so that it gives a single result in which all variable bindings in the projection are replaced with a plain literal, holding the value of the total number of solutions of the query. In the case of a CONSTRUCT query in which the projection contains three variables (`?subject`, `?predicate`, `?object`), the subject and the predicate are bound to `<http://www.ontotext.com/>` and the object holds the literal value. This is because there cannot exist a statement with a literal in the place of the subject or predicate. This clause is deprecated in favor of using the `COUNT` aggregate of SPARQL 1.1.

`FROM` / `FROM NAMED <http://www.ontotext.com/skip-redundant-implicit>`

Used for triggering the exclusion of implicit statements when there is an explicit one within a specific context (even default). Initially implemented to allow for filtering of redundant rows where the context part is not taken into account and which leads to 'duplicate' results.

`FROM <http://www.ontotext.com/distinct>`

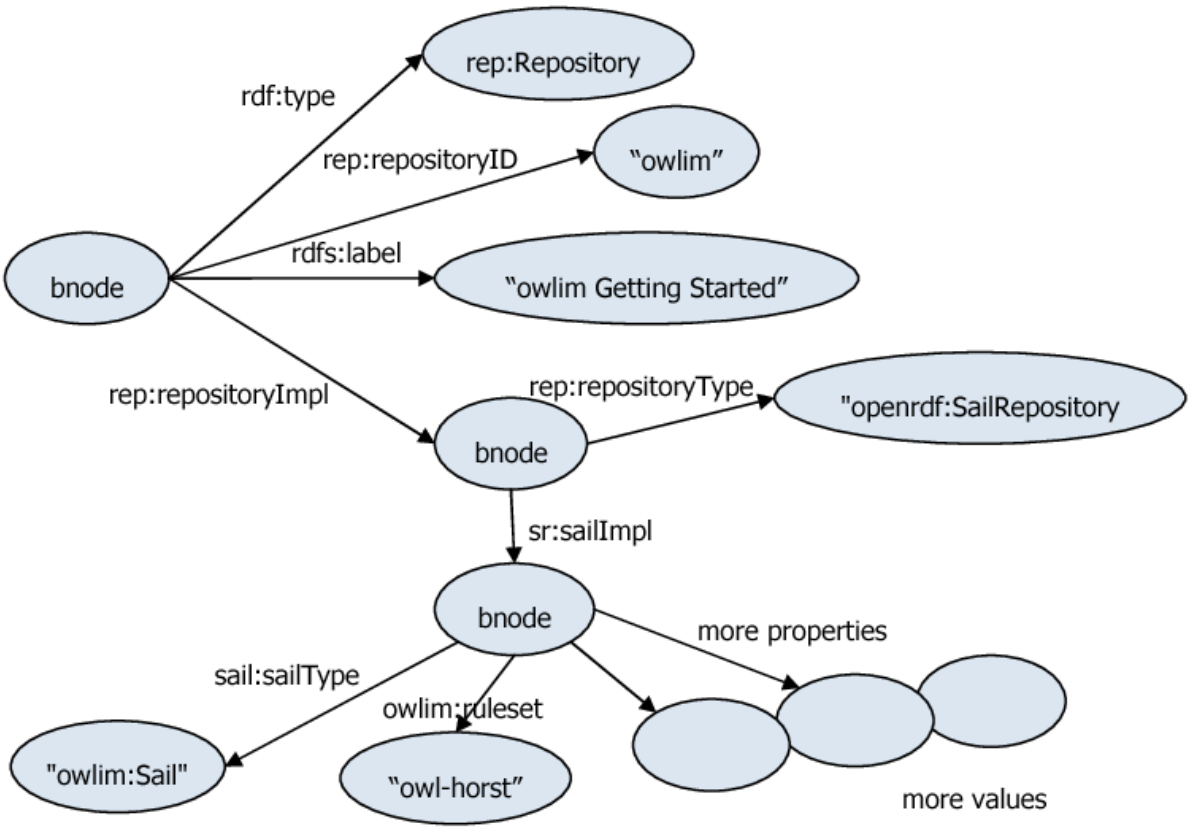
Using this special graph name in `DESCRIBE` and `CONSTRUCT` queries will cause only distinct triples to be returned. This is useful when several resources are being described, where the same triple can be returned more than once, i.e., when describing its subject and its object. This clause is deprecated in favor of using the `DISTINCT` clause of SPARQL 1.1.

`FROM <http://www.ontotext.com/owlim/cluster/control-query>`

Identifies the query to a GraphDB EE cluster master node as needing to be routed to all worker nodes.

7.5.5 How the Repository configuration template works

The diagram below provides an illustration of an RDF graph that describes a repository configuration:



Often, it is helpful to ensure that a repository starts with a predefined set of RDF statements — usually one or more schema graphs. This is possible by using the `graphdb:imports` property. After start-up, these files are parsed and their contents are permanently added to the repository.

In short, the configuration is an RDF graph, where the root node is of `rdf:type rep:Repository`, and it must be connected through the `rep:RepositoryID` property to a Literal that contains the human readable name of the repository. The root node must be connected via the `rep:repositoryImpl` property to a node that describes the configuration.

7.5.5.1 GraphDB repository

> The type of the repository is defined via the `rep:repositoryType` property and its value must be `graphdb:SailRepository` to let RDF4J know what the desired Sail repository implementation is. Then, a node that specifies the Sail implementation to be instantiated must be connected through the `sr:sailImpl` property. To instantiate GraphDB, this last node must have a property `sail:sailType` with the value `graphdb:Sail` — the RDF4J framework will locate the correct `SailFactory` within the application `classpath` that will be used to instantiate the Java implementation class.

The prefixes used above represent the following namespaces:

```
rep: <http://www.openrdf.org/config/repository#>
sr: <http://www.openrdf.org/config/repository/sail#>
sail: <http://www.openrdf.org/config/sail#>
graphdb: <http://www.ontotext.com/trree/graphdb#>
```

All properties used to specify the GraphDB *configuration parameters* use the `graphdb:` prefix and the local names match up with the configuration parameters. For example, the value of the `ruleset` parameter can be specified using the `graphdb:ruleset` property.

7.6 Talk to Your Graph examples

Once you have configured *Talk to Your Graph*, you can create an agent and use it to query your data using natural language. The examples below use two different datasets to demonstrate how to achieve that.

7.6.1 Star Wars example

This example uses data describing Star Wars movies, characters, planets and vehicles, and an agent configured to use SPARQL for querying the database and answering questions.

7.6.1.1 Creating a repository and importing the data

Before you start with Talk to Your Graph, you need to create a repository and import the dataset:

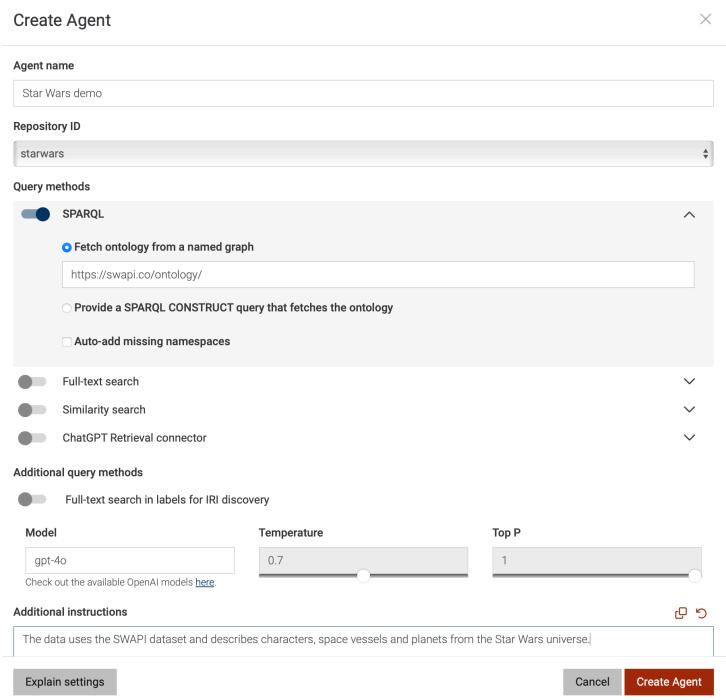
1. First, *create a repository* with a name of your choice (for example, starwars).
2. Then, download the `starwars.trig` dataset and *import* it into your new repository.

7.6.1.2 Talking to your graph

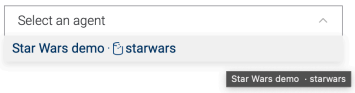
At this point you're set to use the Talk to Your Graph feature. On the Workbench go to *Lab* ▸ *Talk to Your Graph*.

First, you need to create an agent:

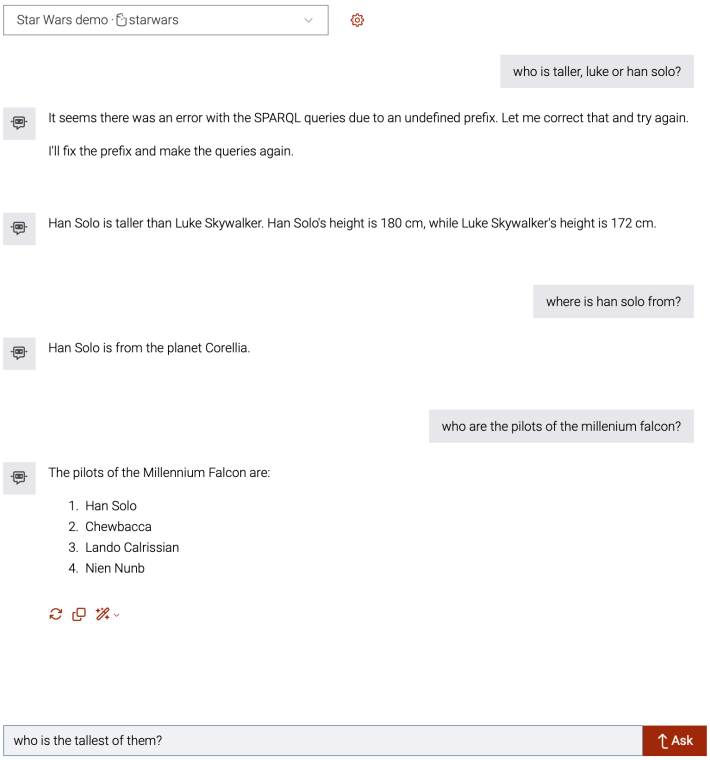
1. Click on the *Create your first agent* button. If you already have one or more agents, click on the *Create a new agent* icon on the top right of the Talk to Your Graph interface. The create agent dialog will open.
2. Enter a name for the agent, for example, "Star Wars demo".
3. Choose the repository you just created (if not already selected).
4. Enable the *SPARQL* query method.
5. Click on *Fetch ontology from a named graph* and enter `https://swapi.co/ontology/` in the named graph field, because that is the ontology named graph in the `starwars.trig` file you downloaded that has the ontology.
6. If needed, scroll down to the *Additional instructions* and enter "The data uses the SWAPI dataset and describes characters, space vessels and planets from the Star Wars universe."
7. Click on the *Create Agent* button.



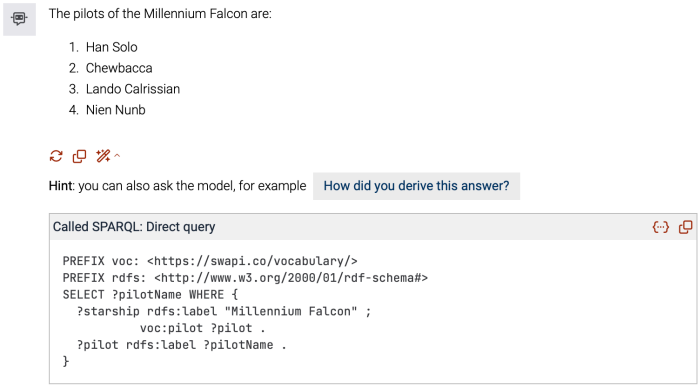
Once the agent has been created, you will see the chat interface. Locate the dropdown field at the top of the screen where it says *Select an agent* and choose the agent you just created. If you already have other agents and chats, the interface will continue showing the last chat and you need to click the *Start a new chat* icon at the top of the chat list. This will initiate a new chat, and then you need to select the agent you created from the dropdown.



Try asking questions about characters in the Star Wars universe:



If you are curious how a particular question was answered, you can click the *Explain response* icon under any response.



7.6.2 Acme employees and products example

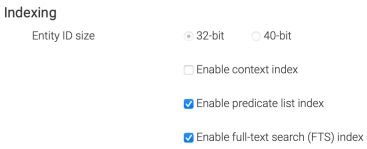
Another dataset and corresponding agent configuration lets you explore the Talk to Your Graph feature with more business-oriented data. It contains data about the employees and products of Acme, an IT company that makes software for the animation industry.

Unlike the Star Wars example, this one will use both SPARQL and full-text search for querying the database.

7.6.2.1 Creating a repository and importing the data

Before you start with Talk to Your Graph, you need to create a repository and import the dataset:

1. First, *create a repository* with a name of your choice (for example, acme). Unlike the previous example, you need to enable the *Simple full-text search index* for this repository by clicking the *Enable full-text search (FTS) index* checkbox in the repository creation form.



2. Then, download the `acme.trig` dataset and *import* it into your new repository.

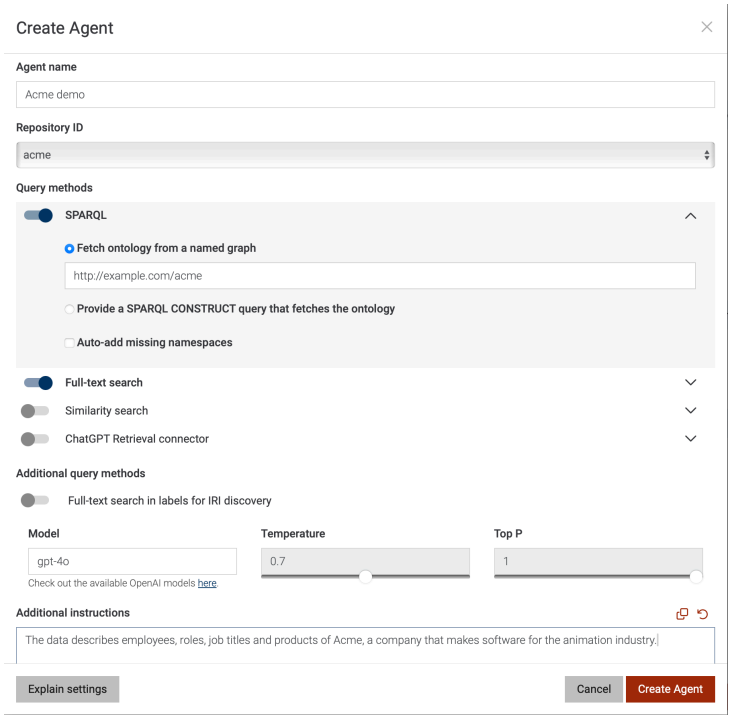
7.6.2.2 Talking to your graph

At this point you're set to use the Talk to Your Graph feature. On the Workbench go to *Lab* ▸ *Talk to Your Graph*.

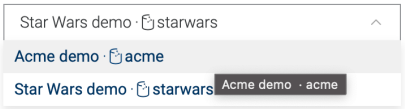
First, you need to create an agent:

1. Click on the *Create your first agent* button. If you already have one or more agents, click on the *Create a new agent* icon on the top right of the Talk to Your Graph interface. The create agent dialog will open.
2. Enter a name for the agent, for example, "Acme demo".
3. Choose the repository you just created if it's not already selected.
4. Enable the *SPARQL* query method.
5. Click on *Fetch ontology from a named graph* and enter `http://example.com/acme` in the named graph field because that is the named graph in the `acme.trig` file you downloaded that has the ontology.
6. Enable the *Full-text search* query method, because we will execute a query that takes advantage of that.

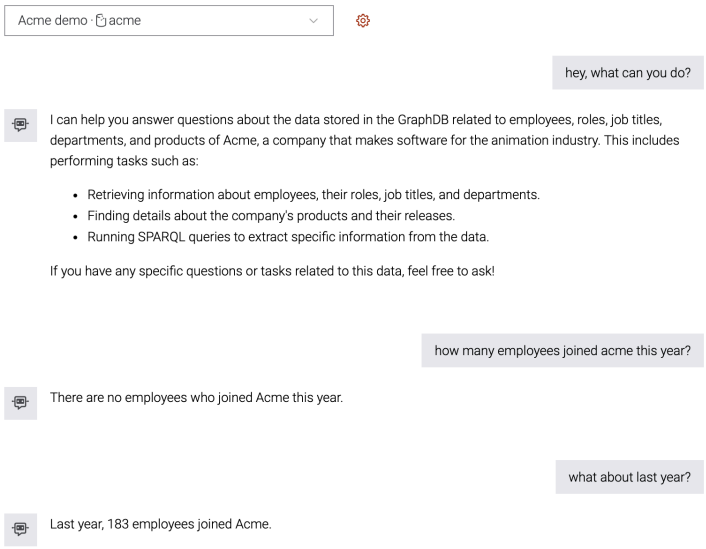
7. If needed, scroll down to the *Additional instructions* and enter "The data describes employees, roles, job titles and products of Acme, a company that makes software for the animation industry."
8. Click on the *Create Agent* button.



Once the agent has been created, you will see the chat interface. Locate the dropdown field at the top of the screen where it says *Select an agent* and choose the agent you just created. If you already have other agents and chats, the interface will continue showing the last chat and you need to click the *Start a new chat* icon at the top of the chat list. This will initiate a new chat, and then you need to select the agent you created from the dropdown.



Try asking questions about employees, departments and products of Acme:



what are acme's products?

Acme's products are:

1. Render Rocket Ultra
2. Plugin Pandemonium Marketplace
3. Charactomatic Extravaganza
4. Motion Mirth Studio

tell me more about motion mirth?

Motion Mirth Studio is a product by Acme described as "Where the real and the animated world collide in a dance of whimsical motion capture magic."

what is the latest version?

The latest version of Motion Mirth Studio is 2.0.

when was that released?

The latest version of Motion Mirth Studio (version 2.0) was released on December 1, 2022.

who has worked at acme the longest?

The employee who has worked at Acme the longest is Stephen P. Jimenez, who started on February 1, 2004.

tell me more about him

Stephen P. Jimenez is the Chief Executive Officer of Acme, a leading software company in the animation industry, where he has dedicated nearly two decades of his career since joining in February 2004. Under his visionary leadership, Acme has launched innovative products that empower animators, including the advanced Render Rocket Ultra and the whimsical Charactomatic Extravaganza, revolutionizing the way animation is created and experienced.

Stephen was born on March 23, 1985, and holds a degree in Computer Science, which laid the foundation for his passion for technology and creativity. Outside of work, he is an avid painter and enjoys hiking, often engaging in outdoor adventures that inspire his artistic endeavors. His commitment to both personal and professional growth has made him a respected figure in the animation software realm.

You can contact him via email at stephen.p.jimenez@example.com.

what does he do at acme?

Stephen P. Jimenez is the Chief Executive Officer (CEO) at Acme, and he works in the Management department.

what about other upper management?

The upper management team at Acme, in addition to Stephen P. Jimenez (Chief Executive Officer), includes:

- Henry Ward, Chief Technology Officer
- Mark Hernandez, Chief Financial Officer
- Deborah Brown, Chief Operating Officer

who is the youngest and the oldest employee?

The youngest employee at Acme is Andrew W. Turner, born on December 29, 1989.

The oldest employee at Acme is Jack O. Parker, born on January 17, 1950.

number of employees per department

Here is the number of employees per department at Acme:

- Management: 4 employees
- Marketing: 76 employees
- Product Development: 881 employees
- Sales: 99 employees

The model answered these questions by using the ontology and writing SPARQL queries. Let's try a more open-ended question:

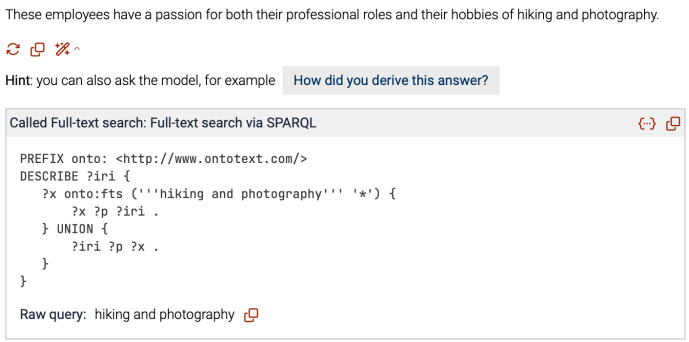
is there anyone at acme who likes hiking and photography?

Yes, there are several employees at Acme who enjoy hiking and photography. Here are a few of them:

1. **Samuel Reyes**
   - Email: samuel.reyes@example.com
   - Biography: Samuel Reyes served as Development Manager at Acme for over six years. Outside of work, he enjoys hiking and photography, capturing the world around him for inspiration in both his personal and professional endeavors.
2. **Janice F. Roberts**
   - Email: janice.f.roberts@example.com
   - Biography: Janice F. Roberts is a seasoned Development Manager at Acme. Outside of work, she enjoys hiking and photography, which inspire her professional projects and fuel her passion for storytelling through visuals.
3. **Emily Davis**
   - Email: emily.davis@example.com
   - Biography: Emily Davis is a dedicated Marketing Specialist at Acme. In her spare time, Emily enjoys experimenting with digital art, participating in local art exhibitions, and exploring the outdoors through hiking and photography.
4. **Adam F. Carter**
   - Email: adam.f.carter@example.com
   - Biography: Adam F. Carter is a talented Software Developer in the Product Development department at Acme. Outside of work, Adam enjoys exploring 3D modeling and animation, as well as hiking and photography.
5. **Martha Campbell**
   - Email: martha.campbell@example.com
   - Biography: Martha Campbell is a dedicated Software Developer at Acme. Outside of work, she enjoys hiking and photography, which enriches her professional insights.

These employees have a passion for both their professional roles and their hobbies of hiking and photography.

If you click the *Explain response* icon under that response you will see the model used the Full-text search query method to answer that question.



Note: Whether the model uses SPARQL or another enabled query method like Full-text search is not strictly predictable. In fact, the question about employees liking hiking and photography could be answered by a SPARQL query that searches the employees' biographies for keywords.

7.7 Vector search in nested fields with Elasticsearch

Semantic vector search over RDF data enables retrieval based on conceptual similarity rather than exact lexical matching. By integrating GraphDB with *Elasticsearch* (as well as OpenSearch) through the connector framework, RDF literals can be automatically transformed into vector embeddings while maintaining their associations to the graph, which allows for high-dimensional search within the context of their structural relationships. This allows SPARQL queries to delegate semantic similarity search to Elasticsearch while still returning RDF resources as results.

A critical modeling decision in such configurations is how multivalued and multilingual properties are indexed. In the Star Wars dataset, both film titles (`rdfs:label`) and descriptions (`https://swapi.co/vocabulary/desc`) are provided in multiple languages. A single film may therefore have several literal values for the same property, each tagged with a different language (for example, English, German, or Finnish).

If these values were indexed without using `datatype: native:nested`, they would be treated as simple multivalued fields. In that scenario, GraphDB would concatenate the literal values into a single logical field during indexing, and embeddings would be generated over the aggregated content rather than per individual literal, eliminating the one-to-one association between a specific language literal and its corresponding embedding vector. Similarity search would then operate over merged multilingual text, reducing precision and making it impossible to distinguish which language-specific value contributed to the similarity score.

By defining the fields as `native:nested`, each literal is stored as a separate nested object which Elasticsearch maintains as independent subdocuments. This guarantees that every language-specific literal is indexed independently and remains structurally bound to its own embedding.

The GraphDB Elasticsearch documentation describes the *Prerequisites for vector search* using the connector, as well as how to perform *Vector search queries*. However, if you want to combine vector search with *Nested objects*, creating a connector that supports both can prove to be more involved than usual. This tutorial covers this more complicated scenario by demonstrating how the GraphDB Elasticsearch connector can be used to enable semantic vector search with nested objects over the Star Wars RDF dataset, specifically films defined in the SWAPI vocabulary.

Tip: Although this example focuses on Elasticsearch, *OpenSearch* also supports vector search, and the example provided here can prove useful when creating an OpenSearch connector with both vector search and nested fields.

7.7.1 Prerequisites for vector search in nested fields

This example uses the Star Wars RDF dataset, which contains data about the Star Wars movies. In this dataset, some of the properties — such as the descriptions of the movies — are present in several different languages, a perfect use case for the application of nested objects in vector search.

1. Create a repository. For the purposes of this tutorial, you can name it `starwars`.
2. Import the `starwars.trig` dataset in the repository.
3. Configure the *embedding model you want to use for vector search*.

7.7.2 Creating the Elasticsearch connector

Create the Elasticsearch connector using the following query:

```
PREFIX conn:<http://www.ontotext.com/connectors/elasticsearch#>
PREFIX inst:<http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
    inst:starwars-index conn:createConnector '''
{
  "fields": [
    {
      "fieldName": "film_label",
      "propertyChain": [
        "http://www.w3.org/2000/01/rdf-schema#label"
      ],
      "datatype": "native:nested",
 "objectFields": [
        {
          "fieldName": "film_label_text",
          "propertyChain": [
            "$self"
          ]
        },
            {
          "fieldName": "film_label_vector",
          "propertyChain": [
            "$self"
          ],
              "datatype": "vector"
        }
      ]
    },
    {
      "fieldName": "film_description",
      "propertyChain": [
        "https://swapi.co/vocabulary/desc"
      ],
      "datatype": "native:nested",
      "objectFields": [
        {
          "fieldName": "film_description_text",
          "propertyChain": [
            "$self"
          ]
        },
            {
          "fieldName": "film_description_vector",
          "propertyChain": [
            "$self"
```

(continues on next page)

(continued from previous page)

```
          ],
              "datatype": "vector"
        }
      ]
    }
  ],
  "types": [
    "https://swapi.co/vocabulary/Film"
  ],
  "embeddingModel" : "com.ontotext.embeddings.GraphwiseTransformerClient",
  "elasticsearchNode": "localhost:9200",
  "languages": [],
  "readonly": false,
  "detectFields": false,
  "importGraph": false,
  "skipInitialIndexing": false,
  "elasticsearchClusterSniff": false,
  "manageIndex": true,
  "manageMapping": true,
  "bulkUpdateBatchSize": 5000
}
''' .
}
```

Note: The `embeddingModel` configuration property in this query assumes the use of the Graphwise Transformer Client, described in the *Embedding model configuration for vector search* documentation.

This will create an Elasticsearch connector instance named `starwars-index` using the `conn:createConnector` predicate, which accepts a JSON configuration describing how RDF data should be indexed and how embeddings should be generated. The connector is configured to index only resources of type `https://swapi.co/vocabulary/Film` — in other words, only entities representing films from the Star Wars dataset will be exported to Elasticsearch.

Two RDF properties are indexed — `rdfs:label` and `voc:desc`. The first corresponds to the films title, and the second to its description. Both fields are defined as `native:nested`, which means they are stored in Elasticsearch as nested objects rather than flat fields.

Inside each nested field definition, two object fields are declared. For the label, these are `film_label_text` and `film_label_vector`. The `$self` property chain indicates that the literal value itself should be used. The `_text` field stores the raw string value of the RDF literal, while the `_vector` field stores its embedding. The embedding is generated automatically because the connector is configured with the embedding model `com.ontotext.embeddings.GraphwiseTransformerClient`. This model converts textual literals into numerical vectors suitable for semantic similarity search. For more information: https://graphdb.ontotext.com/documentation/11.3/embedding-model-configuration.html#using-a-local-embedding-model-deployed-with-graphwise-transformer.

The same structure is used for film_description, where both the textual description and its vector embedding are stored. As a result, each film document in Elasticsearch contains nested objects holding both original text and its corresponding embedding representation.

The remaining configuration parameters define operational behavior. The connector runs against an Elasticsearch node at localhost:9200. Since manageIndex and manageMapping are set to true, the connector automatically creates and manages the Elasticsearch index and mapping. skipInitialIndexing is false, so existing RDF data is indexed immediately after connector creation. The connector is not read-only, meaning it synchronizes changes from GraphDB to Elasticsearch.

7.7.3 Search query for similar label

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX inst: <http://www.ontotext.com/connectors/elasticsearch/instance#>
PREFIX conn: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?entity ?actualLabel {
    ?search a inst:starwars-index ;
        conn:query '''
        {
            "size": 3,
            "query": {
                "nested": {
                    "path": "film_label",
                    "query": {
                        "knn": {
                            "field": "film_label.film_label_vector",
                            "query_vector": "Tähtien varjossa",
                            "k": 5
                        }
                    }
                }
            }
        }
        ''' ;
        conn:entities ?entity .
    ?entity rdfs:label ?actualLabel .
    FILTER(LANG(?actualLabel) = "en")
}
```

Because of the vector fingerprint, you don't have to search for exact words anymore. This query performs semantic similarity search over labels (film titles). Since the indexed labels are stored as nested objects, a nested query must be used inside the Elasticsearch JSON. Within that nested context, a KNN query performs vector similarity search on the field `film_label.film_label_vector`.

The `query_vector` is provided as a text string because the connector uses the configured embedding model. This text is automatically transformed into a vector before executing the k-nearest neighbors search. The parameter `k: 5` specifies that the five closest vectors should be considered, and `size: 3` limits the returned hits to three. The connector then binds the matching RDF entities to ?entity, and their English labels are retrieved via `rdfs:label`.

The query should return the following results:

	entity	actualLabel
1	https://swapi.co/resource/film/4	"Star Wars: Episode I — The Phantom Menace"@en
2	https://swapi.co/resource/film/5	"Star Wars: Episode II — Attack of the Clones"@en
3	https://swapi.co/resource/film/1	"Star Wars"

Note: When the same query is executed using an English translation instead of the Finnish title, the results are almost identical but may differ slightly in ranking. This difference occurs because each language-specific label is embedded and indexed independently — the Finnish query aligns most strongly with Finnish label vectors, while the English query aligns with English label vectors. Since values are not concatenated but stored as separate nested objects, similarity scoring is performed per literal value.

7.7.4 Search query for similar description

The query below applies the same principle to film descriptions. A nested kNN search is performed over `film_description.film_description_vector`. The German query text is embedded and compared against all indexed description vectors.

```
PREFIX voc: <https://swapi.co/vocabulary/>
PREFIX inst: <http://www.ontotext.com/connectors/elasticsearch/instance#>
PREFIX conn: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?entity ?descNoLang {
    ?search a inst:starwars-index ;
        conn:query '''
        {
            "size": 3,
            "query": {
                "nested": {
                    "path": "film_description",
                    "query": {
                        "knn": {
                            "field": "film_description.film_description_vector",
                            "query_vector": "Diese epische Weltraumoper folgt dem eskalierenden Kampf
↪zwischen einem tyrannischen galaktischen Regime und     einer kleinen Rebellengruppe, während ein
↪junger Held in einer alten mystischen Disziplin trainiert und seine engsten     Gefährten von einem
↪mächtigen Vollstrecker des Regimes gnadenlos durch die Galaxie verfolgt werden; während
↪Freundschaften auf    die Probe gestellt werden und schwierige Entscheidungen getroffen werden müssen,
↪ vertieft die Geschichte den Konflikt und   gestaltet das Schicksal ihrer zentralen Figuren neu.",
                            "k": 5
                        }
                    }
                }
            }
        }
        ''' ;
        conn:entities ?entity .
    ?entity voc:desc ?descNoLang .
    FILTER(LANG(?descNoLang) = "")
}
```

The query should provide results such as:

	entity	actualLabel
1	https://swapi.co/resource/film/2	"The Empire Strikes Back (also known as Star Wars: Episode V — The Empire…
2	https://swapi.co/resource/film/1	"Star Wars (later retitled Star Wars: Episode IV — A New Hope) is a 1977 Ame…
3	https://swapi.co/resource/film/5	"Star Wars: Episode II — Attack of the Clones is a 2002 American epic space…

Note: When the same semantic content is queried in English, the most relevant film remains the same, but lower-ranked results may differ in order. This confirms that English and German descriptions are indexed independently and that similarity is computed against each language-specific embedding separately.

See also:

- Elasticsearch's documentation on multi-level nesting
- *Elasticsearch GraphDB Connector* documentation.
- *Embedding model configuration for vector search* documentation.

- *OpenSearch GraphDB Connector* documentation.

CHAPTER

EIGHT

REFERENCE

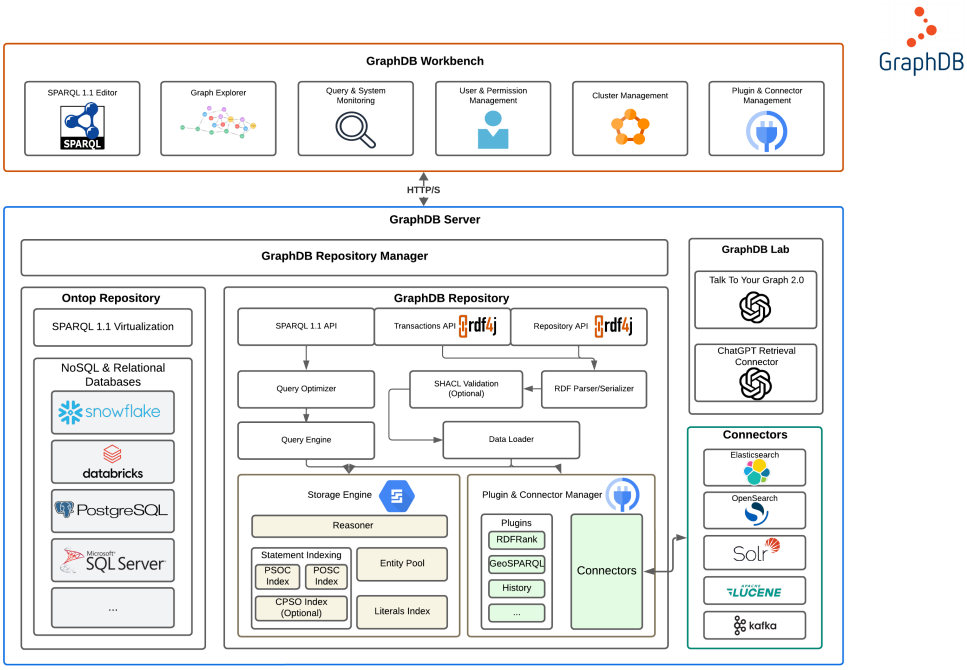
8.1 Architecture and components

8.1.1 Architecture

GraphDB is packaged as a SAIL (Storage And Inference Layer) for the *RDF4J* framework and makes extensive use of the features and infrastructure of RDF4J, especially the *RDF* model, RDF parsers, and query engines.

Inference is performed by *Reasoner (TRREE engine)*, where the explicit and inferred statements are stored in highly optimized data structures that are kept in memory for query evaluation and further inference. The inferred closure is updated through inference at the end of each transaction that modifies the repository.

GraphDB implements *The Sail API* interface so that it can be integrated with the rest of the RDF4J framework, e.g., the query engines and the web UI. A user application can be designed to use GraphDB directly through the RDF4J SAIL API or via the higher-level functional interfaces. When a GraphDB repository is exposed using the RDF4J HTTP Server, users can manage the repository through the embedded *Workbench*, the RDF4J Workbench, or other tools integrated with RDF4J.



8.1.1.1 RDF4J

The RDF4J framework is a framework for storing, querying, and reasoning with RDF data. It is implemented in Java by Aduna as an open-source project and includes various storage back-ends (memory, file, database), query languages, reasoners, and client-server protocols.

There are essentially two ways to use RDF4J:

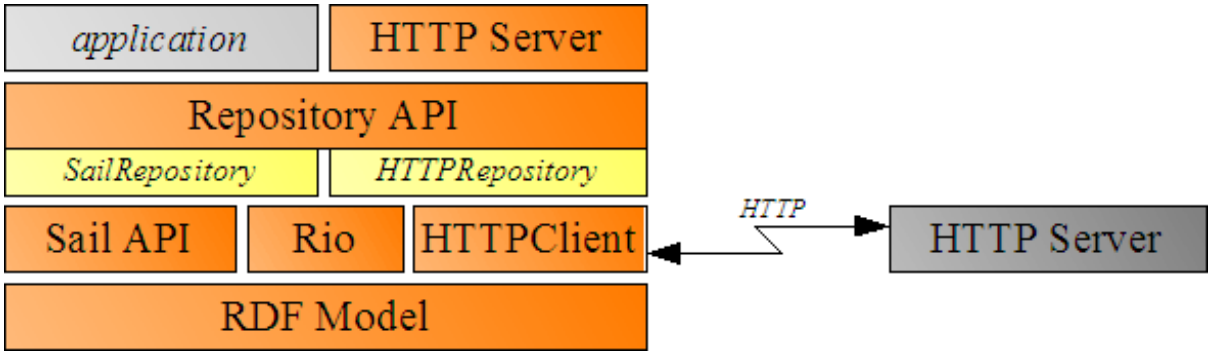
- as a standalone server
- embedded in an application as a Java library

RDF4J supports the W3C SPARQL query language, as well as the most popular RDF file formats and query result formats.

RDF4J offers a JDBC-like user API, streamlined system APIs and a RESTful HTTP interface. Various extensions are available or are being developed by third parties.

RDF4J Architecture

The following is a schematic representation of the RDF4J architecture and a brief overview of the main components.



The RDF4J architecture

The RDF4J framework is a loosely coupled set of components, where alternative implementations can be easily exchanged. RDF4J comes with a variety of SAIL implementations that a user can select for the desired behavior (in-memory storage, file system, relational database, etc). GraphDB is a plugin SAIL component for the RDF4J framework.

Applications will normally communicate with RDF4J through the Repository API. This provides a sufficient level of abstraction so that the details of particular underlying components remain hidden, i.e., different components can be swapped without requiring modification of the application.

The Repository API has several implementations, one of which uses HTTP to communicate with a remote repository that exposes the Repository API via HTTP.

8.1.1.2 The Sail API

The Sail API is a set of Java interfaces that support RDF storage, retrieval, deletion, and inferencing. It is used for abstracting from the actual storage mechanism. For example, an implementation can use relational databases, file systems, or in-memory storage. One of its key characteristics is the option for SAIL stacking.

8.1.2 Components

8.1.2.1 Engine

Query optimizer

The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible *query plans*. Once queries are submitted and parsed, they are then passed to the query optimizer. Learn more in the *Query optimizations* section of GraphDB's Administration documentation.

Reasoner (TRREE engine)

GraphDB is implemented on top of TRREE, or "Triple Reasoning and Rule Entailment Engine". TRREE performs reasoning based on *forward-chaining* of entailment rules over RDF triple patterns with variables. TRREE's reasoning strategy is *total materialization*, although various *optimizations* are used. Further details about the rule language can be found in the *Reasoning* section.

Storage

GraphDB stores all of its data in files in the configured storage directory, usually called *storage*. It consists of two main indexes on statements, `POS` and `PSO`, *context index* `CPSO`, and *literal index*, with the latter two being optional. The spelling of these index names uses initials to reference the role of Contexts, Predicates, Subjects, and Objects in the index keys.

Entity Pool

The Entity Pool is a key component of the GraphDB storage layer. It converts entities (*URIs*, blank nodes, literals, and RDF-star [formerly RDF*] embedded triples) to *internal IDs* (32- or 40-bit integers). It supports transactional behavior, which improves space usage and cluster behavior.

Page cache

GraphDB's *cache strategy* employs the concept of one global cache shared between all internal structures of all repositories, so that you no longer have to configure the `cache-memory`, `tuple-index-memory` and `predicate-memory`, or size every instance and calculate the amount of memory dedicated to it. If one of the repositories is used more at the moment, it naturally gets more slots in the cache.

8.1.2.2 Connectors

The Connectors provide extremely fast keyword and faceted (aggregation) searches that are typically implemented by an external component or service, but have the additional benefit of staying automatically up-to-date with the GraphDB repository data. GraphDB comes with the following connector implementations:

- *Lucene GraphDB Connector*
- *Solr GraphDB Connector* (requires a GraphDB Enterprise license)
- *Elasticsearch GraphDB Connector* (requires a GraphDB Enterprise license)
- *OpenSearch GraphDB Connector* (requires a GraphDB Enterprise license)

The *Kafka GraphDB Connector* provides a means to synchronize changes to the RDF model to any Kafka consumer (requires a GraphDB Enterprise license)

The *ChatGPT Retrieval GraphDB Connector* provides conversion from RDF to text documents and indexing into a vector database through the ChatGPT Retrieval Plugin.

8.1.2.3 Workbench

Workbench is the GraphDB web-based administration tool.

8.2 GraphDB command line tools

The GraphDB distribution includes a number of command line tools located in the `bin` folder of the GraphDB *distribution directory*. Their file extensions are `.sh` or empty for Linux and `.cmd` for Windows.

8.2.1 console

This is an interactive console based on the RDF4J console. After starting it up, entering `help` at its prompt gives you further information about operations that you can perform at that prompt.

Usage: `console [OPTION] [repositoryID]`.

Use `--help` to see the available command line options, which are:

Option	Description
`-c,--cautious`	Always answer no to (suppressed) confirmation prompts
`-d,--dataDir <arg>`	Sesame data directory to 'connect' to
`-e,--echo`	Echoes input back to stdout, useful for logging script sessions
`-f,--force`	Always answer yes to (suppressed) confirmation prompts
`-h,--help`	Print this help
`-q,--quiet`	Suppresses prompts, useful for scripting
`-s,--serverURL <arg>`	URL of Sesame server to connect to, e.g., `http://localhost/openrdf-sesame/`
`-v,--version`	Print version information
`-x,--exitOnError`	Immediately exit the console on the first error

8.2.2 generate-report

This tool is used to generate a zip with report about a GraphDB server. On startup, `graphdb -p` specifies a PID file to which to write the process ID, which is needed by this tool.

Usage: `<graphdb-pid> [<output-file>]`.

The available options are:

Option	Description
`<graphdb-pid>`	(Required) The process ID of a running GraphDB instance.
`<output-file>`	(Optional) The path of the file where the report should be saved. If this option is missing, the report will be saved in a file called `graphdb-server-report.zip` in the current directory.

8.2.3 graphdb

The `graphdb` command line tool starts the database. It supports the following options:

Option	Description
`-d`	daemonize (run in background), not available on Windows
`-s`	run in server-only mode (no Workbench UI)
`-p pidfile`	write PID to `pidfile`
`-h` `--help`	print command line options
`-v`	print GraphDB version, then exit
`-Dprop`	set Java system property
`-Xprop`	set non-standard Java system property

Note: Run `graphdb -s` to start GraphDB in server-only mode without the web interface (no Workbench). A remote Workbench can still be attached to the instance.

8.2.4 importrdf

The `importrdf` tool is used for offline loading of datasets. It supports two sub-commands — Load and Preload.

See more about using this utility in *Loading data using the ImportRDF tool*.

8.2.4.1 Load command line options

Usage: `importrdf load [option] [file]`

Option	Short version	Description
`--config-file <file>`	`-c`	Repository-defining `.ttl` file.
`--force`	`-f`	Whether to overwrite the existing repository.
`--help`	`-h`	Display this message and exit.
`--repository <repository-name>`	`-i`	Name of an existing repository.
`--mode <serial|parallel>`	`-m`	Single-threaded (serial) or multi-threaded (parallel) mode for parse/load/infer.
`--partial-load`	`-p`	Whether to allow partial load of a file that contains a corrupt line.
`--stop-on-error`	`-s`	Whether to stop the process if the dataset contains a corrupt file.
`--verbose`	`-v`	Whether to print metrics during load.

Note: The `--partial-load` will load data up to the first corrupt line of the file.

The mode specifies the way the data is loaded in the repository:

- `serial`: parsing is followed by entity resolution, which is then followed by load, followed by inference, all done in a single thread.
- `parallel`: using multi-threaded parse, entity resolution, load, and inference. This gives a significant boost when loading large datasets with enabled inference.

If no mode is selected, `serial` will be used.

Tip: For loading datasets larger than several billion RDF statements, consider using the Preload sub-command.

8.2.4.2 Preload command line options

Usage: `importrdf preload [option] [file]`

Option	Short version	Description
`--iterator-cache <arg>`	`-a`	Chunk iterator cache size. The value will be multiplied by 1,024. Default is `auto`, e.g., calculated by the tool.
`--chunk <arg>`	`-b`	Chunk size for partial sorting of the queues. Use `m` for millions or `k` for thousands. Default is `auto`, e.g., calculated by the tool.
`--config-file <file>`	`-c`	Repository-defining `.ttl` file.
`--force`	`-f`	Whether to overwrite the existing repository.
`--help`	`-h`	Display this message and exit.
`--id <repository-id>`	`-i`	Existing repository ID.
`--queue-folder <folder>`	`-q`	Folder used to store temporary data.
`--recursive`	`-r`	Whether to walk folders recursively.
`--parsing-tasks <num>`	`-t`	Number of RDF parsers.
`--restart`	`-x`	Whether to restart the load, ignoring any existing recovery points.
`--recovery-point-interval <sec>`	`-y`	The interval at which recovery points are created.

8.2.5 rdfvalidator

Parses a file of RDF data, assuming a serialization based on the filename extension, and reports any errors. If there is no output the input is valid RDF.

If a directory is provided as an argument, this parses all the RDF files in that directory.

Usage: `rdfvalidator <input-folder-or-file-with-rdf-files>`.

8.2.6 reification-convert

This tool converts standard RDF reification to RDF-star. The output file must be an RDF-star format.

Usage: `reification-convert [--relaxed] <input-file1> [<input-file2> ...] <output-file>`.

Available options:

Option	Description
`--relaxed`	Enables relaxed mode where `x a rdf:Statement` is not required.

8.2.7 rule-compiler

Usage: `rule-compiler <rules.pie> <java-class-name> <output-class-file> [<partial>]`.

The `.pie` file format is described more in *Custom rulesets*.

Available options:

Option	Description
`<rules.pie>`	The name of the rule `.pie` file
`<java-class-name>`	The name of the Java class
`<output-class-file>`	The output file name
`[<partial>]`	(Optional)

8.2.8 storage-tool

The `storage-tool` is an application for scanning and repairing a GraphDB repository.

To run it, execute the `bin/storage-tool` script in the GraphDB distribution folder.

For help, run:

```
bin/storage-tool --help
```

Note: The tool works only on repository images that are not in use (i.e., when the database is down).

8.2.8.1 Supported commands

Command	Description
`scan`	Scans repository index(es) and prints statistics for the number of statements and repo consistency.
`rebuild`	Uses the source index (`src-index`) to rebuild the destination index `dest-index`. If `src-index` = `dest-index`, compacts `dest-index`. If `src-index` is missing and `dest-index` = `predicates`. then it just rebuilds `dest-index`.
`replace`	Replaces an existing entity `origin-uri` with a non-existing one `repl-uri`.
`repair`	Repairs the repository indexes and restores data, a better variant of the merge index.
`export`	Uses the source index (`src-index`) to export repository data to the destination file (`dest-file`). Supported destination file extension formats: `.trig`, `.ttl`, `.nq`.
`epool`	Scans the entity pool for inconsistencies and checks for invalid IRIs. IRIs are validated against the RFC 3987 standard. Invalid IRIs will be listed in an `entities.invalid.log` file for review. If `-fix` is specified, instead of listing the invalid IRIs, they will instead be fixed in the entity pool.
`--help`	Prints command-specific help messages.

8.2.8.2 Options

Parameter	Short version	Description	Default value
`--storage`	`-s`	(required) Absolute path to repo storage directory	null
`--help`	`-h`	Prints out help messages	
`--src-index`	`-r`	Predicate collection to be used as source. Can be one of pso, pos.	null
`--dest-index`	`-d`	Predicate collection to be used as destination. Can be one of pso, pos, cpso, predicates.	null
`--origin-uri`	`-o`	Original existing URI in the repository to be replaced	null
`--repl-uri`	`-n`	New non-existing URI in the repository to replace the original	null
`--dest-file`	`-f`	Path to file used to store exported data. Supported formats: `.trig`, `.ttl`, `.nq`.	null
`--fix`	`-x`	Lists or fixes ePool problems.	`false`
`--check-pred-statistics`	`-c`	Runs additional check of predicates statistics	`false`
`--status-print-interval`	`-i`	Interval between status message printing (in seconds)	`30`, means 30 seconds
`--page-cache-size`	`-p`	Size of the page cache (in thousands).	`10`, means 10,000 elements
`--positive-filter-status`	`-v`	Optional statement status filter during export	`-1`, means no filter
`--sort-buffer-size`	`-b`	Size of the external sort buffer	`100`, means 100 million elements, max value is also `100`

8.2.8.3 Examples

- scan the repository, print statement statistics and repository consistency status:

```
bin/storage-tool scan --storage /<path-to-repo>/storage
```

  - when everything is OK

```
    Scan result consistency check!

_______________________scan results_______________________
mask |              pso |              pos | diff | flags
0001 |       29,937,266 |       29,937,266 |   OK | INF
0002 |       61,251,058 |       61,251,058 |   OK | EXP
0005 |              145 |              145 |   OK | INF RO
0006 |            8,134 |            8,134 |   OK | EXP RO
0009 |        1,661,585 |        1,661,585 |   OK | INF HID
000a |        2,834,694 |        2,834,694 |   OK | EXP HID
0011 |        1,601,875 |        1,601,875 |   OK | INF EQ
0012 |        1,934,013 |        1,934,013 |   OK | EXP EQ
0020 |              309 |              221 |   OK | DEL
0021 |               15 |               23 |   OK | INF DEL
0022 |               34 |               30 |   OK | EXP DEL
_______________________additional checks_______________________
     |              pso |              pos | stat | check-type
     |         59b30d4d |         59b30d4d |   OK | checksum
     |                0 |                0 |   OK | not existing ids
     |                0 |                0 |   OK | literals as subjects
```

(continues on next page)

(continued from previous page)

```
    |                0 |                0 |   OK | literals as predicates
    |                0 |                0 |   OK | literals as contexts
    |                0 |                0 |   OK | blanks as predicates
    |             true |             true |   OK | page consistency
    |         80b9ad24 |         80b9ad24 |   OK | cpso crc
    |                - |                - |   OK | epool duplicate ids
    |                - |                - |   OK | epool consistency
    |                - |                - |   OK | literal index consistency
    |                - |                - |   OK | triple entity index consistency

Scan determines that this repo image is consistent.
```

– when there are broken indexes

```
    ________________________scan results________________________
mask |              pso |              pos | diff | flags
0001 |       29,284,580 |       29,284,580 |   OK | INF
0002 |       63,559,252 |       63,559,252 |   OK | EXP
0004 |            8,134 |            8,134 |   OK | RO
0005 |            1,140 |            1,140 |   OK | INF RO
0009 |        1,617,004 |        1,617,004 |   OK | INF HID
000a |        3,068,289 |        3,068,289 |   OK | EXP HID
0011 |        1,599,375 |        1,599,375 |   OK | INF EQ
0012 |        2,167,536 |        2,167,536 |   OK | EXP EQ
0020 |              327 |              254 |   OK | DEL
0021 |               11 |               12 |   OK | INF DEL
0022 |               31 |               24 |   OK | EXP DEL
004a |               17 |               17 |   OK | EXP HID MRK

________________________additional checks________________________
    |              pso |              pos | stat | check-type
    | ffffffff93e6a372 | ffffffff93e6a372 |   OK | checksum
    |                0 |                0 |   OK | not existing ids
    |                0 |                0 |   OK | literals as subjects
    |                0 |                0 |   OK | literals as predicates
    |                0 |                0 |   OK | literals as contexts
    |                0 |                0 |   OK | blanks as predicates
    |             true |             true |   OK | page consistency
    |         bf55ab00 |         bf55ab00 |   OK | cpso crc
    |                - |                - |   OK | epool duplicate ids
    |                - |                - |   OK | epool consistency
    |                - |                - |  ERR | literal index consistency

Scan determines that this repo image is INCONSISTENT.
```

The literals index contains more statements than the literals in epool, and you have to rebuild it:

- scan the `PSO` index and print a status message every 60 seconds:

```
bin/storage-tool scan --storage /<path-to-repo>/storage --src-index=pso --status-print-interval=60
```

- compact the `PSO` index (self-rebuild equals compacting):

```
bin/storage-tool rebuild --storage /<path-to-repo>/storage --src-index=pso --dest-index=pso
```

- rebuild the `POS` index from the `PSO` index and compact `POS`:

```
bin/storage-tool rebuild --storage /<path-to-repo>/storage --src-index=pso --dest-index=pos
```

- rebuild the predicates statistics index:

```
bin/storage-tool rebuild --storage /<path-to-repo>/storage --dest-index=predicates
```

- replace `http://onto.com#e1` with `http://onto.com#e2`:

```
bin/storage-tool replace --storage /<path-to-repo>/storage --origin-uri="<http://onto.com#e1>" --
↪repl-uri="<http://onto.com#e2>"
```

- dump the repository data using the `POS` index into a `f.trig` file:

```
bin/storage-tool export --storage /<path-to-repo>/storage --src-index=pos --dest-file=/repo/
↪storage/f.trig
```

- scan the entity pool and create a report with invalid IRIs, if such exist:

```
bin/storage-tool epool --storage /<path-to-repo>/storage
```

8.3 Directories and configuration properties

GraphDB relies on several main directories for configuration, logging, and data.

8.3.1 Directories

8.3.1.1 GraphDB home

The GraphDB home defines the root directory where GraphDB stores all of its data. The home can be set through the system or config file property `graphdb.home`.

The default value for the GraphDB home directory depends on how you run GraphDB:

- Running as a standalone server: the default is the same as the distribution directory.
- All other types of installations: OS-dependent directory.
  - On Mac: `~/Library/Application Support/GraphDB`.
  - On Windows: `\Users\<username>\AppData\Roaming\GraphDB`.
  - On Linux and other Unixes: `~/.graphdb`.

Note: In the unlikely case of running GraphDB on an ancient Windows XP, the default directory is `\Documents and Settings\<username>\Application Data\GraphDB`.

GraphDB does not store any files directly in the home directory, but uses the following subdirectories for data or configuration:

Data directory

The GraphDB data directory defines where GraphDB stores repository data. The data directory can be set through the system or config property `graphdb.home.data`. The default value is the `data` subdirectory relative to the GraphDB home directory.

Configuration directory

The GraphDB configuration directory defines where GraphDB looks for user-definable configuration. It can be set through the system property `graphdb.home.conf`.

Note: It is not possible to set the config directory through a config property as the value needs to be set before the config properties are loaded.

The default value is the `conf` subdirectory relative to the GraphDB home directory.

Work directory

The GraphDB work directory defines where GraphDB stores non-user-definable configuration. The work directory can be set through the system or config property `graphdb.home.work`. The default value is the `work` subdirectory relative to the GraphDB home directory.

Logs directory

The GraphDB logs directory defines where GraphDB stores log files. The logs directory can be set through the system or config property `graphdb.home.logs`. The default value is the `logs` subdirectory relative to the GraphDB home directory.

Note: When running GraphDB as deployed `.war` files, the logs directory will be a subdirectory `graphdb` within the Tomcat's logs directory.

Tip: Even though GraphDB provides the means to specify separate custom directories for data, configuration and so on, it is recommended to specify the home directory only. This ensures that every piece of data, configuration, or logging, is within the specified location.

Step-by-step guide:

1. Choose a directory for GraphDB home, e.g., `/opt/graphdb-instance`.
2. Create the directory `/opt/graphdb-instance`.
3. (Optional) Copy the subdirectory `conf` from the distribution into `/opt/graphdb-instance`.
4. Start GraphDB with `graphdb -Dgraphdb.home=/opt/graphdb-instance`.

GraphDB creates the missing subdirectories `data`, `conf` (if you skipped that step), `logs`, and `work`.

8.3.1.2 Distribution directory

The distribution directory stores the GraphDB binaries and additional files such as sample data, template files, and command line tools.

When running GraphDB Desktop, the simplest way to find the distribution directory is by clicking the *Open dist directory* button on the *GraphDB Desktop* application window. See *Configuring your running GraphDB instance* for more information on the application window.

When running the GraphDB standalone server (as opposed to the GraphDB Desktop) the distribution directory is the directory where the downloaded distribution file was unzipped.

To learn more about the structure of the distribution directory, see *Structure of the distribution package*.

8.3.1.3 Checking the configured directories

When GraphDB starts, it logs the actual value for each of the above directories, e.g.:

```
GraphDB Home directory: /path-to-graphdb/app
GraphDB Config directory: /path-to-graphdb/app/conf
GraphDB Data directory: /path-to-graphdb/app/data
GraphDB Work directory: /path-to-graphdb/app/work
GraphDB Logs directory: /path-to-graphdb/app/logs
```

8.3.2 Configuration

There is a single `graphdb.properties` config file for GraphDB. It is provided in the distribution under `conf/graphdb.properties`, where GraphDB loads it from.

This file contains a list of config properties defined in the following format:

`propertyName = propertyValue`, i.e., using the standard Java properties file syntax.

Each config property can be overridden through a Java system property with the same name, provided in the environment variable `GDB_JAVA_OPTS`, or in the command line.

8.3.2.1 Storing property values in a separate file

You can also store the value for any configuration property in a separate text file, and point to that file from `graphdb.properties`. The path can be either relative of absolute. Relative paths are relative to the GraphDB `/conf` directory found in the distribution folder.

The text file containing the value must contain a single non-blank line that contains only the property value, although blank lines before or after are allowed. The file has a size limit of 1024 bytes.

Property names pointing to a file end with `@file`. For example, `graphdb.auth.ldap.bind.userDn.password=PASSWORD_VALUE` is how the LDAP bind user password property would appear in `graphdb.properties` (in this example, with a value of `PASSWORD_VALUE`). The same can be achieved by setting `graphdb.auth.ldap.bind.userDn.password@file=/opt/mysecret.txt`, where the file `mysecret.txt` contains that same value `PASSWORD_VALUE` as a single line of text.

This mechanism provides more flexibility to configure your GraphDB instances, as you won't need to configure the individual `graphdb.properties` file for each separate instance. For example, a password or a token can be generated and stored in a separate file, referenced in `graphdb.properties` by using the suffix `@file`, and distributed among all relevant GraphDB instances without editing each individual `graphdb.properties` file.

8.3.2.2 Configuring GraphDB through environment variables

You can also configure GraphDB through the environment variables.

GraphDB reads everything that starts with `graphdb.`, `graphdb_`, `graphdb__` or `graphdb___`. Environment variables configured this way take precedence over what has been configured in `graphdb.properties`.

If you configure GraphDB this way, you can circumvent the limited character set available for environment variables in many systems by using the following special character processing:

	Environment variable example	How it is processed by GraphDB
Period `.`	`graphdb.property`	No changes (`graphdb.property`)
Underscore `_`	`graphdb_property`	No changes (`graphdb_property`)
Double underscores `__`	`graphdb__property`	Processed to `.` (`graphdb.property`)
Triple underscore `__`	`graphdb___property`	Processed to `-` (`graphdb-property`)

Example configurations of GraphDB through externalized environment variables

Configuring GraphDB in Docker

This example shows how to configure GraphDB's GPT settings with environment variables from Docker Compose.

```
version: '3.7'

services:
  graphdb:
    image: ontotext/graphdb:10.7.0
    restart: unless-stopped
    ports:
      - "7201:7201"
    environment:
      GDB_JAVA_OPTS: -Xms4G -Xmx4G
      graphdb.connector.port: "7201"
      graphdb.llm.api-key: "<secret-token>"
    volumes:
      - graphdb_data:/opt/graphdb/home

volumes:
  graphdb_data:
```

Configuring OIDC in Kubernetes

Create a configuration map with the following data and provide it as environment to the pod or container:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: graphdb-oidc
data:
  graphdb.auth.methods: gdb,openid
  graphdb.auth.openid.issuer: https://login.microsoftonline.com/[ID]/v2.0
  graphdb.auth.openid.client_id: [CLIENT-ID]
  graphdb.auth.openid.username_claim: email
  graphdb.auth.openid.auth_flow: code
  graphdb.auth.openid.token_type: id
  graphdb.auth.database: local
  graphdb.auth.oauth.roles_claim: roles
  graphdb.auth.oauth.roles_prefix: GDB_
  graphdb.auth.oauth.default_roles: ROLE_USER
```

And then use it in GraphDB's pods with:

```
envFrom:
  - configMapRef:
      name: graphdb-oidc
```

Changing the cluster secret in Kubernetes

This example highlights why using a Secret object is better than using `GDB_JAVA_OPTS` when running GraphDB in Kubernetes. Usually, such sensitive configurations are created beforehand and have to be provided as environment, or even mounted and read as files.

Create a secret with the following content and provide it as environment.

```
apiVersion: v1
kind: Secret
metadata:
  name: graphdb-cluster-secret
data:
  graphdb.auth.token.secret: [YOUR-SECRET]
```

And then use it in a pod:

```
envFrom:
- secretRef:
    name: graphdb-cluster-secret
```

8.3.2.3 Configuration properties

There are seven types of properties described below: General properties, Workbench properties, URL properties, Cluster properties, GraphQL properties, Network properties, and Engine properties.

General properties

The general properties define some basic configuration values that are shared with all GraphDB components and types of installation:

Property name	Description
`graphdb.home`	Defines the GraphDB home directory
`graphdb.home.data`	Defines the GraphDB `data` directory
`graphdb.home.conf`	(only as a system property) Defines the GraphDB `conf` directory
`graphdb.home.work`	Defines the GraphDB `work` directory
`graphdb.home.logs`	Defines the GraphDB `logs` directory
`graphdb.dist`	If `graphdb.dist` is set and `graphdb.home` is not, GraphDB will look for the `data`, `conf`, `logs`, etc. directories there (unless they are explicitly set).
`graphdb.workbench.home`	The place where the source for GraphDB Workbench is located
`graphdb.jsonld.whitelist`	Sets whitelist for JSON-LD and NDJSON-LD resources necessary for importing and exporting data to certain *JSON-LD or NDJSON-LD document forms*.
`graphdb.license.file`	Sets a custom path to the license file to use
`graphdb.page.cache.size`	The amount of memory to be taken by the page cache
`graphdb.pidfile`	The full path to the file where the GraphDB process ID is stored
`graphdb.foreground`	Tells GraphDB not to close `stdout/stderr`, but the user can choose whether to daemonize or not
`graphdb.heapdump.enable`	GraphDB can dump the heap on out-of-memory errors in order to provide insight to the cause for excessive memory usage. This property enables or disables the heap dump. Default is `true`.
`graphdb.heapdump.path`	File to write the heap dump to. The default is the `heapdump.hprof` file in the configured `logs` directory. See also the properties `graphdb.home` and `graphdb.home.logs`.
`graphdb.max.active.repositories`	Used to define the maximum number of active repositories, as described in the *Limiting the number of active repositories* section of the *Repository Caching Mechanism* documentation. Setting this property to less than 1 (preferably 0) will deactivate the mechanism. The default is `100`.
`graphdb.repository.expiry.after.minutes`	*Sets a timeout period* in minutes after which a repository that has not been accessed will be shut down by the *Repository Caching Mechanism*. A value of `0` disables the mechanism. The default is `0`.
`graphdb.inference.buffer`	Buffer size (the number of statements) for each load stage in *parallel import*. Defaults to 200,000 statements. See also `graphdb.inference.concurrency`.
`graphdb.inference.concurrency`	Number of inference threads in *parallel import*. The default value is the number of cores of the machine processor. See also `graphdb.inference.buffer`.
`graphdb.ontop.jdbc.path`	GraphDB directory for the JDBC driver used in the creation of Ontop repositories. Use it when you want to set it to a directory different from the `lib/jdbc` one where the driver is normally placed.
`graphdb.fts.hard-limit`	Defines the absolute maximum of results that can be returned by a single *FTS search*. Default size is `10000`.

Workbench properties

In addition to the standard GraphDB command line parameters, the GraphDB Workbench can be controlled with the following parameters (they should be of the form `-Dparam=value`):

Parameter	Description	Default value
`graphdb.workbench.cors.enable`	Enables cross-origin resource sharing.	`false`
`graphdb.workbench.cors.origin`	Sets the allowed Origin value for cross-origin resource sharing. This can be a comma-delimited list or a single value. The value "*" means "allow all origins" and it works with authentication too.	`*`

`graphdb.workbench.cors.expose-headers`	As per GraphDB's compliance with the Access-Control-Expose-Headers, when the two parameters above are enabled, this parameter exposes headers other than the CORS-safelisted request headers. They are exposed in a comma-delimited list.
	
	Example:
	`graphdb.workbench.cors.enable=true`
	
	`graphdb.workbench.cors.origin=*`
	
	`graphdb.workbench.cors.expose-headers="content,location"`
		If no value is set, only the CORS-safelisted request headers will be exposed.

`graphdb.workbench.maxConnections`	Sets the maximum number of concurrent connections to a remote GraphDB instance.	`200`
`graphdb.workbench.datadir`	Sets the directory where the workbench persistence data will be stored.	`${user.home}/.graphdb-workbench/`
`graphdb.workbench.importDirectory`	Changes the location of the file import folder.	`${user.home}/graphdb-import/`
`graphdb.workbench.maxUploadSize`	Sets the maximum upload size for importing local files. The value must be in bytes.	1 gb

URL properties

Tip: Jump ahead to *Typical use cases* for a list of examples that cover URL properties usage.

In certain cases, GraphDB needs to construct a URL that refers to itself:

- The repository list in *Setup ▸ Repository manager* where each repository provides a link that can be used to access the repository via the REST API.

When GraphDB is accessed directly (without a reverse proxy), it will figure out the correct URLs based on the URL of incoming requests. For example, if GraphDB is accessed using the URL `http://graphdb.example.com:7200/`, it will construct URLs like `http://graphdb.example.com:7200/repositories/repoId`.

When GraphDB is accessed via a reverse proxy, the server will not see the actual URL used to access the server and thus it cannot determine a valid external URL on its own. There are two specific setups:

- The external URL as seen via the proxy uses `/` as its root, for example, `http://rdf.example.com/`.

  - GraphDB will map the external `/` to its own `/` automatically, no need to add or change any configuration.
  - GraphDB will still not know how to construct external URLs, so setting `graphdb.external-url` is recommended even though it might appear to work without setting it.
- The external URL as seen via the proxy uses `/something` as its root (i.e., something in addition to the `/`), for example, `http://example.com/rdf`.
  - GraphDB cannot map this automatically and needs to be configured using the property `graphdb.vhosts` or `graphdb.external-url` (see below).
  - This will instruct GraphDB that URLs beginning with `http://example.com/rdf/` map to the root path `/` of the GraphDB server.

The URL properties determine how GraphDB constructs URLs that refer to itself, as well as what URLs are recognized as URLs to access the GraphDB installation. GraphDB will try to auto-detect those values based on URLs used to access it, and the network configuration of the machine running GraphDB. In certain setups involving virtualization or a reverse proxy, it may be necessary to set one or more of the following properties:

Property	Description
`graphdb.vhosts`	A comma-delimited list of virtual host URLs that can be used to access GraphDB. Setting this property is necessary when GraphDB needs to be accessed behind a reverse proxy and the path of the external URL is different from `/`, for example `http://example.com/rdf`.
`graphdb.external-url`	Sets the canonical external URL. This property implies `graphdb.vhosts`. If you have provided an explicit value for both `graphdb.vhosts` and `graphdb.external-url`, then the URL specified for `graphdb.external-url` must be one of the URLs in the value for `graphdb.vhosts`. When a reverse proxy is in use and most users will access GraphDB through the proxy, it is recommended to set this property instead of, or in addition to `graphdb.vhosts`, as it will let GraphDB know that the canonical external URL is the one as seen through the proxy. **Tip:** Prior to GraphDB 9.8, only the `graphdb.external-url` property existed. You can keep using it as is.
`graphdb.external-url.enforce.transactions`	Determines whether it is necessary to rewrite the Location header when no proxy is configured. Setting this property to `true` will use the `graphdb.external-url` when building the transaction URLs. Set it to `true` when the returned URLs are incorrect due to missing or invalid proxy configurations. Set it to `false` when the server can be called on multiple addresses, as it will override the returned address to the one defined by the `graphdb.external-url`. Boolean, default is `false`.
`graphdb.hostname`	Overrides the hostname reported by the machine.

Enabling the configuration will use the graphdb.external-url when building the transaction URLs. It should be used when the returned URLs are not correct due to missing or invalid proxy configurations. The configuration should not be used when the server can be called on multiple addresses as it will override the returned address to a single one defined by the graphdb.external-url.

Note: For remote locations, the URLs are always constructed using the base URL of the remote location as specified when the location was attached.

Typical use cases

1. GraphDB is behind a reverse proxy whose URL path is `/` and most clients will use the proxy URL.

   This setup will appear to work out-of-the box without setting any of the URL properties but it is recommended to set `graphdb.external-url`. Example URLs:

   - Internal URL: `http://graphdb.example.com:7200/`
   - External URL used by most clients: `http://rdf.example.com/`

   The corresponding configuration is:

   ```
   # Recommended even though it may appear to work without setting this property
   graphdb.external-url = http://rdf.example.com/
   ```

2. GraphDB is behind a reverse proxy whose URL path is `/something` and most clients will use the proxy URL.

   This configuration requires setting `graphdb.external-url` (recommended) or `graphdb.vhosts` to the correct URLs as seen externally through the proxy. Example URLs:

   - Internal URL: `http://graphdb.example.com:7200/`
   - External URL used by most clients: `http://example.com/rdf/`

   The corresponding configuration is:

   ```
   # Required and recommended
   graphdb.external-url = http://example.com/rdf/

   # Non-recommended alternative to the above
   #graphdb.vhosts = http://example.com/rdf/
   ```

Cluster properties

Parameter	Description	Default value
`graphdb.cluster.sync.timeoutS`	Specifies the maximum duration in seconds that the system waits for the completion of cluster creation before timing out.	`600` (10 minutes)
`graphdb.cluster.catchup.snapshot.replication.timeoutS`	Specifies the maximum duration in seconds that the system waits to replicate the cluster data on a new node before timing out.	`10800` (3 hours)
`graphdb.cluster.catchup.follower.sync.timeoutS`	Specifies the maximum duration in seconds that the system waits for a follower node to catch up with the latest cluster data and get in sync with the other nodes before timing out.	`1800` (30 minutes)
`graphdb.cluster.proxy.socketTimeout`	Specifies the maximum duration in seconds a node waits for various operations (such as connecting to a server, reading data from the socket, or writing data to the socket after establishing a connection with another node in the cluster group) before timing out. Part of custom GraphDB HTTP client configurations representing defaults for request redirection.	`-1` (indefinite)
`graphdb.cluster.proxy.maxConnectionsTotal`	Defines the default maximum number of concurrent connections that a node can establish to other nodes within the cluster group. Part of custom GraphDB HTTP client configurations.	`50000`
`graphdb.cluster.proxy.maxConnectionsPerRoute`	Defines the default maximum number of concurrent connections that a node can establish to another node within the cluster group. Part of custom GraphDB HTTP client configurations.	`30000`
`graphdb.cluster.proxy.connectionTimeoutS`	Specifies the maximum duration in seconds a node is willing to wait to establish a connection to another node in the cluster group before timing out. Part of custom GraphDB HTTP client configurations.	`15`
`graphdb.cluster.proxy.socket.soTimeout`	Specifies the maximum time in milliseconds a node is willing to wait to read data after establishing a connection with another node in the cluster group before timing out. Part of custom GraphDB HTTP client configurations.	`18000000` (5 hours) if TLS is enabled, otherwise `0` indicating no timeout (which defaults to 30 seconds).
`graphdb.cluster.tls.trust.own.certificate`	Determines whether the cluster should accept the provided SSL certificate as a trusted one.	`TRUE`
`graphdb.cluster.node.report.staleAfterSeconds`	Specifies the time in seconds after which a new system report will be triggered upon receiving a customer request, provided the specified time has elapsed since the previous report was made.	`60`
`graphdb.cluster.observers.autoUnregisterAfterMinutes`	Specifies the time in minutes after which an observer is automatically unregistered if it has not received an update within that timeframe.	`15`

The raft properties below control the security configuration for the gRPC communications.

Parameter	Description	Example value
`graphdb.raft.security.mode`	Determines the security mode configuration. **Warning:** If set to `TLS` while one or more of the other TLS-related properties are not configured properly, the server may not be able to start.	`DEFAULT`, `TLS`, `NONE`
`graphdb.raft.security.certificateRevocationListFile`	Determines a list of digital certificates that have been revoked by the issuing certificate authority (CA) before their actual or assigned expiration date.	
`graphdb.raft.security.certificateVerification`	Determines the settings for certificate verification during security communication.	`required`, `optional`, `optional-NoCA`
`graphdb.raft.security.certificateChainFile`	Specifies the file path or location where the SSL/TLS certificate chain file is stored. The format is PEM-encoded.	
`graphdb.raft.security.certificateFile`	Specifies the file path or location where the SSL/TLS certificate file is stored. The format is PEM-encoded.	
`graphdb.raft.security.certificateKeyFile`	Specifies the file path or location where the private key associated with the SSL/TLS certificate is stored. The format is PEM-encoded.	
`graphdb.raft.security.certificateKeyPassword`	Specifies the password or passphrase used to protect the private key associated with the SSL/TLS certificate.	
`graphdb.raft.security.algorithm`	Specifies a particular algorithm or configuration parameter within the	

GraphQL properties

The GraphQL properties control how the GraphQL mechanism operates in GraphDB. Most of these properties correspond to similar properties in Semantic Objects, which have been listed in the table below for the purposes of migrating from Semantic Objects to GraphDB.

Parameter	Description	Default value
`graphdb.graphql.query.optimizations.mutationMode`	Specifies the write mode to the underlying GraphDB repository. The possible values are: • `default`: Placeholder for the application default. The default value. • `read_write`: Modifications will affect the existing data in the repository. By default, all data will be written to the default graph, but this parameter also allows writing in a custom graph passed in the mutation request. Default behavior. • `changes`: Modifications will affect the existing data in the repository. All data inserts will be done in either per-entity graphs or in a custom graph passed in the mutation request. • `read_only`: Modifications will not be possible and will always fail. **Corresponding Semantic Objects property:** `sparql.optimizations.mutationMode`	`default`
`graphdb.graphql.query.optimizations.disableUnionToLateralOptimization`	Controls if union optimization should be disabled. The union optimization includes how SPARQL is generated for multi-valued properties in the presence of filters in the GraphQL queries. When enabled, lateral nested selects will be used to fetch multi-valued properties instead of SPARQL UNION. **Corresponding Semantic Objects property:** `sparql.optimizations.disableUnionToLateralOptimization`	`false`
`graphdb.graphql.response.compactErrorMessages`	Controls how the GraphQL Responder processes errors that are returned to the client. When `true`, it will force the GraphQL Responder to override most of the returned messages and to return a single compacted one for each of the following cases: lack of permissions, missing required data, multiple values for single-valued properties and unexpected or bad data format. This could be overridden by adding `compactErrorMessages: true` in the config section of the SOML or by passing the `errorsFormat: FULL` argument to a GraphQL query or mutation. **Corresponding Semantic Objects property:** `graphql.response`	`false`

Network properties

The network properties control how the standalone application listens on a network. These properties correspond to the attributes of the embedded Tomcat Connector. Each property is composed of the prefix `graphdb.connector.` + the relevant Tomcat Connector attribute.

Certain SSL properties are composed slightly differently, taking the prefix `graphdb.connector.ssl.` + the relevant Tomcat Connector SSL attribute:

- `graphdb.connector.ssl.certificateKeyAlias`
- `graphdb.connector.ssl.certificateKeyPassword`
- `graphdb.connector.ssl.certificateKeystoreFile`
- `graphdb.connector.ssl.certificateKeystorePassword`
- `graphdb.connector.ssl.certificateKeystoreProvider`
- `graphdb.connector.ssl.certificateKeystoreType`
- `graphdb.connector.ssl.certificateFile`
- `graphdb.connector.ssl.certificateKeyFile`
- `graphdb.connector.ssl.certificateKeyPassword`
- `graphdb.connector.ssl.truststoreFile`
- `graphdb.connector.sll.truststorePassword`

In addition to those properties, the property `graphdb.connector.ssl.enabled` does not correspond to a single Tomcat Connector attribute, which allows you to enable SSL easily without requiring several different properties.

See also:

- For more information on SSL network properties, see Tomcat's documentation on SSL support
- For information on all other network properties, see Tomcat's documentation on the HTTP connector configurations

The most important property is `graphdb.connector.port`, which defines the port to be used. The default is 7200.

In addition, the sample config file provides an example for setting up SSL.

Note: The `graphdb.connector.<xxx>` properties are only relevant when running GraphDB as a standalone application.

Engine properties

You can configure the GraphDB Engine through a set of properties composed of the prefix `graphdb.engine.` + the relevant engine property. These properties correspond to the properties that can be set when creating a repository through the Workbench or through a `.ttl` file.

Note: The properties defined in the config **override** the properties for each repository, regardless of whether you created the repository before or after setting the global value of an engine property. As such, the global override should be used only in specific cases. For normal everyday needs, set the corresponding properties when you create a repository.

Property name	Description	Default value
`graphdb.engine.entity-pool-implementation`	Defines the Entity Pool implementation for the whole installation. Possible values are `transactional` or `classic`.	The default value is `transactional`. The `transactional-simple` implementation is not supported anymore.
`graphdb.persistent.parallel.inferencers`	Since GraphDB 8.6.1, inferencers for our Parallel loader are shut down at the end of each transaction to minimize GraphDB's memory footprint. For cases where a lot of small insertions are done in a quick succession that can be a problem, as inferencer initialization times can be fairly slow. This setting reverts to the old behavior where inferencers are only shut down when the repository is released.	`false`
`graphdb.engine.entity.validate`	A global setting that ensures IRI validation in the entity pool. It is performed only when an IRI is seen for the first time (i.e., when being created in the entity pool). For consistency reasons, not only IRIs coming from RDF serializations, but also all new IRIs (via API or SPARQL), will be validated in the same way. This property can be turned off by setting its value to `false`.	`true`
`graphdb.health.minimal.free.storage.warn`	Defines the percentage of free disk space that triggers a warning that GraphDB is running low on disk space.	`20`
`graphdb.health.minimal.free.storage.error`	Defines the percentage of free disk space that triggers an error that GraphDB is running low on disk space.	`10`
`graphdb.health.minimal.free.storage.fatal`	Defines the percentage of free disk space that causes GraphDB to prevents new transactions from being started and log a fatal state error.	`5`
`graphdb.health.minimal.free.storage.enabled`	Enables or disables disk space health checks when the repository is initiated and before a transaction begins.	`true`
`graphdb.health.minimal.free.storage.asyncCheck`	Enables or disables asynchronous disk space health checks during writing every 5 seconds by default — controlled by `graphdb.health.minimal.free.storage.asyncCheck.intervalMS` — and additionally makes checks on every 10,000 statements by default, controlled by `graphdb.health.continues.disk.check.operations`.	`true`
`graphdb.health.minimal.free.storage.asyncCheck.intervalMS`	Controls the number of seconds a disk space health check is triggered if `graphdb.health.minimal.free.storage.asyncCheck` is set to true.	`5`

Warning: To disable the low disk space health check mechanism, set both `graphdb.health.minimal.free.storage.enabled` and `graphdb.health.minimal.free.storage.asyncCheck` to `false`. The *Low disk space health checks* page provides more information on the validation checks and the different log messages.

Note: Note that IRI validation makes the import of broken data more problematic - in such a case, you would have to change a config property and restart your GraphDB instance instead of changing the setting per import.

8.3.2.4 Talk to Your Graph and LLM properties

These Large language model (LLM) properties are used to configure the LLM used by various mechanisms in GraphDB, such as *Talk to Your Graph*.

LLM properties

The `graphdb.llm` properties are used to configure the use of Large language models (LLM).

Property name	Description	Default value
`graphdb.llm.api`	The API for the LLM type. Possible values include: • `openai-assistants` • `openai-completions` • `openai-completions-http1.1` • `gemini` • `bedrock` • `vertex`	`openai-assistants`
`graphdb.llm.api-key`	The authentication token for the LLM API.	
`graphdb.llm.url`	The base URL for the LLM provider.	
`graphdb.llm.auth`	The authentication method to use for the chat completions endpoint. Optional. This property does not need to be configured for Azure OpenAI as it is detected automatically. The possible values are: • `auto`: Automatically determines the authentication method. • `bearer`: Sends the token via the HTTP header `Authorization: Bearer <token>`. • `api-key`: Used only with Azure OpenAI. Sends the token via the HTTP header `api-key: <token>`. • `custom`: Sends a token that must consist of a header and value separated by a colon (for example, `my-header:my-auth-value`). GraphDB will send it as the HTTP header `my-header: my-auth-value`. • `none`: No authentication headers will be sent.	`auto`
`graphdb.llm.model`	Determines the default model used when creating new agents in Talk to Your Graph. Also configures the model to use for SPARQL functions and SPARQL explain. **Note:** If the *GPT SPARQL function and SPARQL explain property* `graphdb.gpt-sparql.model` has been set, it will override the model defined by this property when using GPT SPARQL functions and SPARQL explain, but not for Talk to Your Graph.	

Which properties are required depends on the API defined for `graphdb.llm.api`. The table below describes which properties are required for different values of `graphdb.llm.api`.

LLM provider API (`graphdb.llm.api`)	**Base URL** (`graphdb.llm.url`)	**API key** (`graphdb.llm.api-key`)	**Model** (`graphdb.llm.model`)
`openai-assistants`	Default is `OpenAI`, not required	Required	Default is `gpt-4o`, not required
`openai-completions`	Default is `OpenAI`, not required	Required	Required
`openai-completions-http1.1`	Default is `OpenAI`, not required	Required	Required
`gemini`	`gemini` is default	Required	Required

Note: If your provider does not require an API key, set `graphdb.llm.api-key` to `none`.

Tip: If you are not using Large language models with GraphDB, you do not need to set any of these properties.

Google Vertex and Amazon Bedrock configuration properties

There are several LLM configuration properties specific to AWS Bedrock and Google Vertex that start with `graphdb.llm.bedrock` and `graphdb.llm.vortex` respectively.

For Google Vertex, the project ID and the credentials file can also be set through the Application Default Credentials mechanism, if not set by the dedicated properties.

For Amazon Bedrock, `secretAccessKey` and `accessKeyId` can be set by the DefaultCredentialProvider if not set by the dedicated properties, and the region can be set by DefaultAwsRegionProviderChain if not set by `graphdb.llm.bedrock.region`.

For both Amazon Bedrock and Google Vertex, if you want to use a profile different from the default provider, you will need to use the properties below.

Warning: GraphDB will throw an exception if the API is set but is unable to resolve any of the properties below.

Table 1: Google Vertex configuration properties

Property name	Note
`graphdb.llm.vertex.location`	This property is always required when using Google Vertex.
`graphdb.llm.vertex.project.id`	Optional
`graphdb.llm.vertex.credentials`	Optional

Table 2: Amazon Bedrock configuration properties

Property name	Note
`graphdb.llm.bedrock.secretAccessKey`	Optional
`graphdb.llm.bedrock.accessKeyId`	Optional
`graphdb.llm.bedrock.region`	Optional

Talk to Your Graph properties

The `graphdb.ttyg` properties control how the Talk to Your Graph mechanism and its agents operate.

Property name	Description
`graphdb.ttyg.installation.id`	Installation ID for the GraphDB instance. This property is relevant only when using the same LLM project or OpenAI Azure deployment across multiple GraphDB installations, and can be set to a unique value to prevent the different GraphDB instances from seeing each other's agents. The default value is `__default__` and allows agents and chats created with that value to be seen by the same GraphDB instance even if you change it to a custom value at a later point. Default value: `__default__`
`graphdb.ttyg.timeout`	The timeout in seconds when calling the LLM API for Talk to Your Graph only. Default value: `90`
`graphdb.ttyg.base.instructions`	The default base instructions for the agents. Changing the default value is not recommended unless with a very specific intent. Default value: *You are Quadro, an assistant developed by Ontotext and you can answer questions about data stored in GraphDB.*
`graphdb.ttyg.sparql.instructions`	The default SPARQL queries instructions for the agents. Used only if SPARQL search is enabled. Default value: *If you need to write a SPARQL query, use only the classes and properties provided in the schema and don't invent or guess any. Always try to return human-readable names or labels and not only the IRIs. If SPARQL fails to provide the necessary information you can try another tool too.*
`graphdb.ttyg.ontology.introduction`	Default ontology introduction instructions for the agents. Used only if SPARQL search is enabled. Default value: *The ontology schema to use in SPARQL queries is:*
`graphdb.ttyg.http.timeout.connect`	HTTP connect timeout in milliseconds to use in HTTP requests to external services. Currently, only ChatGPT Retrieval uses HTTP connections. Default value: `30000`
`graphdb.ttyg.http.timeout.socket`	HTTP socket timeout in milliseconds to use in HTTP requests to external services. Currently, only ChatGPT Retrieval uses HTTP connections. Default value: `30000`
`graphdb.ttyg.chat.messages.limit`	The max number of messages shown for each chat. Default value: `100`

`graphdb.ttyg.fts.query.template`	The SPARQL template for the full-text search query method.
	Default value:
	`PREFIX onto: <http://www.ontotext.com/>`
	`DESCRIBE ?iri {`
	`  ?x onto:fts (%s '*') {`
	`  ?x ?p ?iri .`
	`  } UNION {`
	`  ?iri ?p ?x .`
	`  }`
	`}`
	Where `%s` is replaced by the value generated from the model.

`graphdb.ttyg.similarity.query.template`	The SPARQL template for the semantic similarity search query method. The value of the first `%s` is replaced by the name of the similarity index itself, as defined by the user. The value of the second `%s` is replaced by the value generated from the model. The value of `%f` is replaced by the similarity score threshold, as defined by the user. The value of this property applies to the whole project not only for the particular agent. When introducing changes to the template, make sure to preserve the order of the name of the similarity index itself (the first `%s`), and the value generated from the model (second `%s`). Failing to do so will break the query method. We also recommend against changing the three placeholders manually. Default value:

	`PREFIX sim: <http://www.ontotext.com/graphdb/similarity/>`
	`PREFIX sim-index: <http://www.ontotext.com/graphdb/similarity/instance/>`
	`DESCRIBE ?documentID WHERE {`
	`  ?search a sim-index:%s ;`

OpenAI configuration properties

There are also two properties for additional configurations when using OpenAI's models. They start with `graphdb.openai`.

`graphdb.openai.project`	Configures the OpenAI project via the OpenAI-Project header to specify which project to use.	
`graphdb.openai.organization`	Configures the OpenAI organization via the OpenAI-Project header to specify which organization to use.	

Note: Previous versions of GraphDB only supported OpenAI's Assistants API and there were more OpenAI configuration properties. Starting from GraphDB 11.1, these properties have been deprecated and replaced with more generic LLM properties. Check the *migration guide for GraphDB 11.1* for a full list of the deprecated properties and their corresponding replacements.

MCP server properties

A set with properties that begins with the prefix `graphdb.mcp.server` and are used to manage session lifecycle and eviction of the Model Context Protocol (MCP) server when *Using GraphDB's LLM tools with MCP clients*.

Property name	Description	Default value
`graphdb.mcp.server.max.sessions`	Maximum number of concurrent MCP sessions allowed at the same time. Once the limit is reached, the server evicts the most idle sessions.	`100`
`graphdb.mcp.server.sessions.keepalive.interval`	Interval (in milliseconds) for sending keep-alive messages to active MCP sessions. This helps prevent idle connections from being closed by intermediate proxies. If the client does not actively manage connection lifecycles, this property should be left unset, so that GraphDB can automatically close idle sessions.	`0`
`graphdb.mcp.server.idle.sessions.timeout`	Timeout (in minutes) after which an idle MCP session will be removed to free server resources facilitate the clean up stale or abandoned sessions and free server resources. A session is considered idle if there has been no activity (incoming messages or keep-alive traffic) for this duration.	`30`
`graphdb.mcp.transport.mode`	The transport mechanism used by the GraphDB MCP server for client communication. The property supports the following values: • `streamable`: The Streamable HTTP transport as defined in the MCP specification. This transport uses HTTP `POST` and `GET` requests for all server-to-client JSON-RPC communication. This mode replaces the older HTTP+SSE transport mechanism. • `sse`: The legacy HTTP + Server-Sent Events (SSE) transport for server-to-client streaming. This mode is supported for backwards compatibility with older clients.	`streamable`

SPARQL functions and SPARQL explain

A set of properties that begin with the prefix `graphdb.gpt-sparql` and are used to configure the use of the *GPT SPARQL functions and SPARQL explain extension*.

Property name	Description	Default value
`graphdb.gpt-sparql.model`	Configures the model or Azure OpenAI model deployment ID to use for SPARQL functions and SPARQL explain. When used for Azure OpenAI, it is used to determine the model deployment ID for the full URL to the Chat Completions API endpoint.	`gpt-4o`
`graphdb.gpt-sparql.api-version`	Configures the API version for Azure OpenAI. The default value is `2024-06-01` and it may change in the future. Unless you need a specific API version, it's recommended to leave this property unset.	`2024-06-01`
`graphdb.gpt-sparql.timeout`	Configures the timeout in seconds when calling the LLM API for SPARQL functions and SPARQL explain only.	`90`

Embedding model properties

The Embedding Model Clients, which are used with *Elasticsearch GraphDB Connector* and *OpenSearch GraphDB Connector*, can read a set of system properties defined through the GraphDB configuration file. These properties can configured through GraphDB, but are described in the configuration section of the Embedding Model Clients project.

8.3.2.5 Configuring logging

GraphDB uses logback to configure logging. The default configuration is provided as `logback.xml` in the GraphDB `conf` directory.

8.4 RDF formats

GraphDB supports multiple RDF formats for importing or exporting data. All RDF formats have at least one file extension and MIME type that identify the format. Where multiple file extensions or MIME types are available, the preferred file extension or MIME type is listed first.

The various formats differ when it comes to supporting named graphs, namespaces, and *RDF-star*. The following formats support everything and may be used to dump an entire repository preserving all of the information:

- *TriG-star* (text, human readable, standard based)
- *BinaryRDF* (binary, compact representation, RDF4J-specific)

8.4.1 Turtle

Named graphs	No
Namespaces	Yes
RDF-star	No
MIME types	`text/turtle` `application/x-turtle`
File extensions	`.ttl`
RDF4J Java API constant	`RDFFormat.TURTLE`
Standard definition	http://www.w3.org/ns/formats/Turtle

8.4.2 Turtle-star

Named graphs	No
Namespaces	Yes
RDF-star	Yes
MIME types	`text/x-turtlestar` `application/x-turtlestar`
File extensions	`.ttls`
RDF4J Java API constant	`RDFFormat.TURTLESTAR`
Standard definition	-

8.4.3 TriG

Named graphs	Yes
Namespaces	Yes
RDF-star	No
MIME types	`application/trig` `application/x-trig`
File extensions	`.trig`
RDF4J Java API constant	`RDFFormat.TRIG`
Standard definition	http://www.w3.org/ns/formats/TriG

8.4.4 TriG-star

Named graphs	Yes
Namespaces	Yes
RDF-star	Yes
MIME types	`application/x-trigstar`
File extensions	`.trigs`
RDF4J Java API constant	`RDFFormat.TRIGSTAR`
Standard definition	-

8.4.5 N3

Named graphs	No
Namespaces	Yes
RDF-star	No
MIME types	`text/n3` `text/rdf+n3`
File extensions	`.n3`
RDF4J Java API constant	`RDFFormat.N3`
Standard definition	http://www.w3.org/ns/formats/N3

8.4.6 N-Triples

Named graphs	No
Namespaces	No
RDF-star	No
MIME types	`application/n-triples` `text/plain`
File extensions	`.nt`
RDF4J Java API constant	`RDFFormat.NTRIPLES`
Standard definition	http://www.w3.org/ns/formats/N-Triples

8.4.7 N-Quads

Named graphs	Yes
Namespaces	No
RDF-star	No

MIME types	`application/n-quads`
	`text/x-nquads`
	`text/nquads`

File extensions	`.nq`
RDF4J Java API constant	`RDFFormat.NQUADS`
Standard definition	http://www.w3.org/ns/formats/N-Quads

8.4.8 JSON-LD

Named graphs	Yes
Namespaces	Yes
RDF-star	No
MIME types	`application/ld+json`
File extensions	`.jsonld`
RDF4J Java API constant	`RDFFormat.JSONLD`
Standard definition	http://www.w3.org/ns/formats/JSON-LD

8.4.9 NDJSON-LD

Named graphs	Yes
Namespaces	Yes
RDF-star	No
MIME types	`application/x-ld+ndjson`

File extensions	`.ndjsonld`
	`.jsonl`
	`.ndjson`

RDF4J Java API constant	`RDFFormat.NDJSONLD`
Standard definition	-

8.4.10 RDF/JSON

Named graphs	Yes
Namespaces	No
RDF-star	No
MIME types	`application/rdf+json`
File extensions	`.rj`
RDF4J Java API constant	`RDFFormat.RDFJSON`
Standard definition	http://www.w3.org/ns/formats/RDF_JSON

8.4.11 RDF/XML

Named graphs	No
Namespaces	Yes
RDF-star	No

MIME types	`application/rdf+xml`
	`application/xml`
	`text/xml`

File extensions	`.rdf`
	`.rdfs`
	`.owl`
	`.xml`

RDF4J Java API constant	`RDFFormat.RDFXML`
Standard definition	http://www.w3.org/ns/formats/RDF_XML

8.4.12 TriX

Named graphs	Yes
Namespaces	Yes
RDF-star	No
MIME types	`application/trix`
File extensions	`.xml` `.trix`
RDF4J Java API constant	`RDFFormat.TRIX`
Standard definition	-

8.4.13 BinaryRDF

Named graphs	Yes
Namespaces	Yes
RDF-star	Yes
MIME types	`application/x-binary-rdf`
File extensions	`.brf`
RDF4J Java API constant	`RDFFormat.BINARY`
Standard definition	https://rdf4j.org/documentation/reference/rdf4j-binary/

8.5 SPARQL compliance

GraphDB supports the following SPARQL specifications:

8.5.1 SPARQL 1.1 Protocol for RDF

SPARQL 1.1 Protocol for RDF defines the means for transmitting SPARQL queries to a SPARQL query processing service, and returning the query results to the entity that requested them.

8.5.2 SPARQL 1.1 Query

SPARQL 1.1 Query provides more powerful query constructions compared to SPARQL 1.0. It adds:

- Aggregates;
- Subqueries;
- Negation;
- Expressions in the SELECT clause;
- Property Paths;
- Assignment;
- An expanded set of functions and operators.

8.5.3 SPARQL 1.1 Update

SPARQL 1.1 Update provides a means to change the state of the database using a query-like syntax. SPARQL Update has similarities to `SQL INSERT INTO`, `UPDATE WHERE`, and `DELETE FROM` behavior. For full details, see the W3C SPARQL Update working group page.

8.5.3.1 Modification operations on RDF triples

- `INSERT DATA {...}`: Inserts RDF statements;
- `DELETE DATA {...}`: Removes RDF statements;
- `DELETE {...} INSERT {...} WHERE {...}`: For more complex modifications;
- `LOAD (SILENT) from_iri`: Loads an RDF document identified by `from_iri`;
- `LOAD (SILENT) from_iri INTO GRAPH to_iri`: Loads an RDF document into the local graph called `to_iri`;
- `CLEAR (SILENT) GRAPH iri`: Removes all triples from the graph identified by `iri`;
- `CLEAR (SILENT) DEFAULT`: Removes all triples from the default graph;

- `CLEAR (SILENT) NAMED`: Removes all triples from all named graphs;
- `CLEAR (SILENT) ALL`: Removes all triples from all graphs.

8.5.3.2 Operations for managing graphs

- `CREATE`: Creates a new graph in stores that support empty graphs;
- `DROP`: Removes a graph and all of its contents;
- `COPY`: Modifies a graph to contain a copy of another;
- `MOVE`: Moves all of the data from one graph into another;
- `ADD`: Reproduces all data from one graph into another.

8.5.4 SPARQL 1.1 Federation

SPARQL 1.1 Federation provides extensions to the query syntax for executing distributed queries over any number of SPARQL endpoints. This feature is very powerful, and allows integration of RDF data from different sources using a single query. See more about it on this documentation's *SPARQL Federation* page.

8.5.4.1 Internal SPARQL Federation

In addition to the standard SPARQL 1.1 Federation to other SPARQL endpoints, GraphDB supports internal federation to other repositories in the same GraphDB instance. The internal SPARQL federation is used in almost the same way as the standard SPARQL federation over HTTP, but since this approach skips all HTTP communication overheads, it is more efficient. See more about it in the *Internal SPARQL federation* section of this documentation's *SPARQL Federation* page.

8.5.4.2 Federated query to a remote password-protected repository

You can also use federation to query a remote password-protected GraphDB repository and a SPARQL endpoint. See how in *Federated query to a remote password-protected repository*.

8.5.5 SPARQL 1.1 Graph Store HTTP Protocol

SPARQL 1.1 Graph Store HTTP Protocol provides a means for updating and fetching RDF graph content from a Graph Store over HTTP in the REST style.

URL patterns for using the Graph Store HTTP Protocol are provided at:

- `<RDF4J_URL>/repositories/<repo_id>/rdf-graphs/service>` (for indirectly referenced named graphs)

8.5.5.1 URL patterns

- `<RDF4J_URL>/repositories/<repo_id>/rdf-graphs/service>` (for indirectly referenced named graphs);
- `<RDF4J_URL>/repositories/<repo_id>/rdf-graphs/<NAME>` (for directly referenced named graphs).

8.5.5.2 Methods supported by these resources and their effects

- `GET`: Fetches statements in the named graph from the repository in the requested format.
- `PUT`: Updates data in the named graph in the repository, replacing any existing data in the named graph with the supplied data. The data supplied with this request is expected to contain an RDF document in one of the *supported RDF formats*.
- `DELETE`: Deletes all data in the specified named graph in the repository.
- `POST`: Updates data in the named graph in the repository by adding the supplied data to any existing data in the named graph. The data supplied with this request is expected to contain an RDF document in one of the *supported RDF formats*.

8.5.5.3 Request headers

- `Accept`: Relevant values for `GET` requests are the MIME types of *the supported RDF formats*.
- `Content-Type`: Must specify the encoding of any request data sent to a server. Relevant values are the `MIME` types of *the supported RDF formats*.

8.5.5.4 Supported parameters for requests on indirectly referenced named graphs

- `graph` (optional): Specifies the URI of the named graph to be accessed.
- `default` (optional): Specifies that the default graph to be accessed. This parameter is expected to be present but to have no value.

Note: Each request on an indirectly referenced graph needs to specify precisely one of the above parameters.

8.5.6 SPARQL 1.1 non-conformance

While GraphDB mostly conforms to the SPARQL 1.1 standard, there are some deviations.

Conformant behavior	Implemented behavior	Reasoning
Any `FILTER` expressions inside a `UNION` should only see bindings created within their branch.	GraphDB allows `FILTER` expressions to evaluate variables that were bound in outer or sibling groups.	Intentional deviation. This behavior provides a more intuitive experience when writing queries where the same variable name is reused. It avoids surprises when `FILTER(?z = ...)` appears to "do nothing" because `?z` was re-bound in a sub-group. The issue can be easily avoided by using unique variable names in nested expressions.
If `SILENT` is used, the engine should ignore any errors coming from the service (such as the remote service not being available), and return local results.	When `SILENT` is used, GraphDB ignores the fact that the remote service is not available, but returns empty results instead of giving only the local data in the results.	Intentional deviation. When users are querying data from a remote service, they most likely need the data from that service specifically. Returning the local data can thus be misleading. **Note:** This behavior can be circumvented by wrapping the service expression with `OPTIONAL` in order to fetch local data.
A `DELETE` that does not specify a graph should only delete data in the default graph.	When `DELETE` is used without specifying a graph, GraphDB deletes data from all graphs. **Warning:** This will be addressed in a future major release of GraphDB (version 12.0 or later), making the behavior conform	Unintentional deviation. However, since there are queries relying on this behavior, and it cannot be fixed in a way that makes queries backwards compatible, it has been intentionally left as is for the time being.

8.5.6.1 Deviations from SPARQL 1.1 tests

While the list above describes all known cases of non-conformance with SPARQL 1.1, GraphDB may also fail to conform to some of the official tests designed to evaluate conformance to the standard. However, for these tests it was determined that there were issues with the test itself, and not with the standard itself. Please note that the list below is not extensive, as there may be other cases of test failures despite GraphDB being compliant with the SPARQL 1.1 specifications.

- syntax-service-01.rq: This test checks the handling of full IRIs without a scheme. GraphDB expects a full IRI, including the scheme (`http://` or `https://`), which fails the test.

  > In the SPARQL standard there are mentions of a nonexistent section 5.1.4, "Default Base URI" which seems to have been deleted from the final specification. It was probably meant to describe how IRIs such as `<g>` should be treated with a `BASE URI`. The test is likely invalid as the official W3C SPARQL 1.1 standard does not explicitly specify how the scheme of such an IRI has to be handled.

- bind07.rq: This test checks the behavior of a `UNION` of two `BIND` expressions so that each binds a different string to `?z`. The test expects the `UNION` to be evaluated first in isolation, then the two `BIND` functions, producing two rows with different `?z` values. In GraphDB, `?x :p ?y` is joined with the entire `UNION`, then the two `BIND` expressions are applied afterwards, causing an unexpected presence of `?z` and resulting in a different binding set than what the test expects.

  > Earlier versions of the SPARQL 1.1 standard had a section that specified how extension functions can be defined and used. That part of the specification was subsequently removed. The current specification does define evaluation order for joins — it just defines what solutions must be produced, and that variable bindings must be respected. This test does not match the current specification.

8.6 SPARQL functions reference

This section lists all supported SPARQL functions in GraphDB. The function specifications include the types of the arguments and the output. Types from XML Schema should be readily recognizable as they start with the `xsd:` prefix. In addition, the following more generic types are used:

`rdfTerm` Any RDF value: a literal, a blank node or an IRI.

`iri` An IRI.

`bnode` A blank node.

`literal` A literal regardless of its datatype or the presence of a language tag.

`string` A plain literal or a literal with a language tag. Note that plain literals have the implicit datatype `xsd:string`.

`numeric` A literal with a numeric XSD datatype, e.g. `xsd:double` and `xsd:long`.

`variable` A SPARQL variable.

`expression` A SPARQL expression that may use any constants and variables to compute a value.

8.6.1 SPARQL 1.1 functions

Function	Description
`xsd:boolean BOUND(variable var)`	Returns `true` if the variable *var* is bound to a value. Returns `false` otherwise. Variables with the value NaN or INF are considered bound. More
`rdfTerm IF(expression e1, expression e2, expression e3)`	The IF function form evaluates the first argument, interprets it as a effective boolean value, then returns the value of *e2* if the EBV is true, otherwise it returns the value of *e3*. Only one of *e2* and *e3* is evaluated. If evaluating the first argument raises an error, then an error is raised for the evaluation of the IF expression. More
`rdfTerm COALESCE(expression e1, …)`	The COALESCE function form returns the RDF term value of the first expression that evaluates without error. In SPARQL, evaluating an unbound variable raises an error. If none of the arguments evaluates to an RDF term, an error is raised. If no expressions are evaluated without error, an error is raised. More

`xsd:boolean NOT EXISTS { pattern }`
 	
`xsd:boolean EXISTS { pattern }`
	There is a filter operator EXISTS that takes a graph pattern. EXISTS returns `true` or `false` depending on whether the pattern matches the dataset given the bindings in the current group graph pattern, the dataset and the active graph at this point in the query evaluation. No additional binding of variables occurs. The NOT EXISTS form translates into `fn:not(EXISTS{...})`. More

`xsd:boolean xsd:boolean left || xsd:boolean right`	Returns a logical OR of *left* and *right*. Note that logical-or operates on the effective boolean value of its arguments. More
`xsd:boolean xsd:boolean left && xsd:boolean right`	Returns a logical AND of *left* and *right*. Note that logical-and operates on the effective boolean value of its arguments. More
`xsd:boolean rdfTerm term1 = rdfTerm term2`	Returns `true` if *term1* and *term2* are equal. Returns `false` otherwise. IRIs and blank nodes are equal if they are the same RDF term as defined in RDF Concepts. Literals are equal if they have an XSD datatype, the same language tag (if any) and their values produced by applying the lexical-to-value mapping of their datatypes are also equal. If the arguments are both literal but their datatype is not an XSD datatype an error will be produced. More
`xsd:boolean sameTerm(rdfTerm term1, rdfTerm term2)`	Returns `true` if term1 and term2 are the same RDF term as defined in RDF Concepts; returns `false` otherwise. More
`xsd:boolean rdfTerm term IN (expression e1, …)`	The IN operator tests whether the RDF term on the left-hand side is found in the values of list of expressions on the right-hand side. The test is done with `=` operator, which compares the RDF term to each expression for equality. More
`xsd:boolean rdfTerm term NOT IN (expression e1, …)`	The NOT IN operator tests whether the RDF term on the left-hand side is not found in the values of list of expressions on the right-hand side. The test is done with `!=` operator, which compares the RDF term to each expression for inequality. More

`xsd:boolean isIRI(rdfTerm term)`
 	
`xsd:boolean isURI(rdfTerm term)`
	Returns `true` if *term* is an IRI. Returns `false` otherwise. More

`xsd:boolean isBlank(rdfTerm term)`	Returns `true` if *term* is a blank node. Returns `false` otherwise. More
`xsd:boolean isLiteral(rdfTerm term)`	Returns `true` if *term* is a literal. Returns `false` otherwise. More

Continued on next page

Table 3 – continued from previous page

Function	Description
`xsd:boolean isNumeric(rdfTerm term)`	Returns `true` if *term* is a numeric value. Returns `false` otherwise. A term is numeric if it has an appropriate datatype and has a valid lexical form, making it a valid argument to functions and operators taking numeric arguments. More

`xsd:string STR(literal ltrl)`
 	
`xsd:string STR(iri rsrc)`
	Returns the lexical form of *ltrl* (a literal); returns the codepoint representation of *rsrc* (an IRI). This is useful for examining parts of an IRI, for instance, the hostname. More

`xsd:string LANG(literal ltrl)`	Returns the language tag of the literal *ltrl*, if it has one. It returns "" if *ltrl* has no language tag. Note that the RDF data model does not include literals with an empty language tag. More
`iri DATATYPE(literal ltrl)`	Returns the datatype IRI of the literal *ltrl*. More

`iri IRI(string str)`
 	
`iri IRI(iri rsrc)`
 	
`iri URI(string str)`
 	
`iri URI(iri rsrc)`
	The IRI function constructs an IRI by resolving the string argument *str*. The IRI is resolved against the base IRI of the query and must result in an absolute IRI. If the function is passed an IRI *rsrc*, it returns the IRI unchanged. More

`bnode BNODE()`
 	
`bnode BNODE(string str)`
	The BNODE function constructs a blank node that is distinct from all blank nodes in the dataset being queried and distinct from all blank nodes created by calls to this constructor for other query solutions. If the no argument form is used, every call results in a distinct blank node. If the form with the string *str* is used, every call results in distinct blank nodes for different strings, and the same blank node for calls with the same string within expressions for one solution mapping. More

`iri UUID()`	Return a fresh IRI from the UUID URN scheme. Each call of UUID() returns a different UUID. More
`xsd:string STRUUID()`	Return a string that is the scheme specific part of UUID. That is, as a string literal, the result of generating a UUID, converting to a string literal and removing the initial `urn:uuid:`. More
`xsd:integer STRLEN(string str)`	The STRLEN function corresponds to the XPath `fn:string-length` function and returns an `xsd:integer` equal to the length in characters of the lexical form of the string *str*. More

`string SUBSTR(string source, xsd:integer startingLoc)`
 	
`string SUBSTR(string source, xsd:integer startingLoc, xsd:integer length)`
	The SUBSTR function corresponds to the XPath `fn:substring` function and returns a literal of the same kind (string literal or literal with language tag) as the *source* input parameter but with a lexical form formed from the substring of the lexical form of the source. More

`string UCASE(string str)`	The UCASE function corresponds to the XPath `fn:upper-case` function. It returns a string literal whose lexical form is the upper case of the lexical form of the argument. More

Continued on next page

Table 3 – continued from previous page

Function	Description
`string LCASE(string str)`	The LCASE function corresponds to the XPath `fn:lower-case` function. It returns a string literal whose lexical form is the lower case of the lexical form of the argument. More
`xsd:boolean STRSTARTS(string str1, string str2)`	The STRSTARTS function corresponds to the XPath `fn:starts-with` function and returns `true` if the lexical form of *str1* starts with the lexical form of *str2*, otherwise it returns `false`. More
`xsd:boolean STRENDS(string str1, string str2)`	The STRENDS function corresponds to the XPath `fn:ends-with` function and returns `true` if the lexical form of *str1* ends with the lexical form of *str2*, otherwise it returns `false`. More
`xsd:boolean CONTAINS(string str1, string str2)`	The CONTAINS function corresponds to the XPath `fn:contains` function and returns `true` if the lexical form of *str1* contains the lexical form of *str2* as a substring. More
`string STRBEFORE(string str1, string str2)`	The STRBEFORE function corresponds to the XPath `fn:substring-before` function and returns a literal of the same kind as *str1*. The lexical form of the result is the substring of the lexical form of *str1* that precedes the first occurrence of the lexical form of *str2*. If the lexical form of *str2* is the empty string, this is considered to be a match and the lexical form of the result is the empty string. If there is no such occurrence, an empty string literal is returned. More
`string STRAFTER(string str1, string str2)`	The STRAFTER function corresponds to the XPath `fn:substring-after` function and returns a literal of the same kind as *str1*. The lexical form of the result is the substring of the lexical form of *str1* that follows the first occurrence of the lexical form of *str2*. If the lexical form of *str2* is the empty string, this is considered to be a match and the lexical form of the result is the empty string. If there is no such occurrence, an empty simple literal is returned. More
`xsd:string ENCODE_FOR_URI(string str)`	The ENCODE_FOR_URI function corresponds to the XPath `fn:encode-for-uri` function. It returns a simple literal with the lexical form obtained from the lexical form of its input after translating reserved characters according to the `fn:encode-for-uri` function. More
`string CONCAT(string str1, …)`	The CONCAT function corresponds to the XPath `fn:concat` function. The function accepts string literals as arguments. The lexical form of the returned literal is obtained by concatenating the lexical forms of its inputs. If all input literals are literals with identical language tag, then the returned literal is a literal with the same language tag, in all other cases, the returned literal is a simple literal. More
`xsd:boolean langMatches(xsd:string languageTag, xsd:string languageRange)`	Returns `true` if *languageTag* (first argument) matches *languageRange* (second argument). According to language tag semantics, the matching is case-insensitive. *languageRange* is a basic language range. For example, "en" will match any of the *languageTags* "en", "EN", "En", "en-GB", "en-US", etc. A language range of "*" matches any non-empty language tag string. More

`xsd:boolean REGEX(string text, xsd:string pattern)`
 	
`xsd:boolean REGEX(string text, xsd:string pattern, xsd:string flags)`
	Invokes the XPath `fn:matches` function to match text against a regular expression pattern. Regular expression matching may involve the modifier `flags`: "i" requests case-insensitive matching. More

Continued on next page

Table 3 – continued from previous page

Function	Description

`string REPLACE(string arg, xsd:string pattern, xsd:string replacement)`
 	
`string REPLACE(string arg, xsd:string pattern, xsd:string replacement, xsd:string flags)`
	The `REPLACE` function corresponds to the XPath `fn:replace` function. It replaces each non-overlapping occurrence of the regular expression pattern with the replacement string. Regular expression matching may involve the modifier `flags`: "i" requests case-insensitive matching. More

`numeric ABS(numeric num)`	Returns the absolute value of *num*. An error is raised if the argument is not a numeric value. This function is the same as `fn:numeric-abs` for terms with a datatype from the XQuery and XPath Data Model specification. More
`numeric ROUND(numeric num)`	Returns the number with no fractional part that is closest to *num*. If there are two such numbers, then the one that is closest to positive infinity is returned. An error is raised if the argument is not a numeric value. This function is the same as `fn:numeric-round` for terms with a datatype from the XQuery and XPath Data Model specification. More
`numeric CEIL(numeric num)`	Returns the smallest (closest to negative infinity) number with no fractional part that is not less than the value of *num*. An error is raised if the argument is not a numeric value. This function is the same as `fn:numeric-ceil` for terms with a datatype from the XQuery and XPath Data Model specification. More
`numeric FLOOR(numeric num)`	Returns the largest (closest to positive infinity) number with no fractional part that is not greater than the value of *num*. An error is raised if the argument is not a numeric value. This function is the same as `fn:numeric-floor` for terms with a datatype from the XQuery and XPath Data Model specification. More
`xsd:double RAND()`	Returns a pseudo-random number between 0 (inclusive) and 1.0 (exclusive). Different numbers can be produced every time this function is invoked. Numbers should be produced with approximately equal probability. More
`xsd:dateTime NOW()`	Returns an XSD dateTime value for the current query execution. All calls to this function in any one query execution will return the same value. The exact moment returned is not specified. More
`xsd:integer YEAR(xsd:dateTime arg)`	Returns the year part of *arg* as an integer. This function corresponds to `fn:year-from-dateTime`. More
`xsd:integer MONTH(xsd:dateTime arg)`	Returns the month part of *arg* as an integer. This function corresponds to *fn:month-from-dateTime*. More
`xsd:integer DAY(xsd:dateTime arg)`	Returns the day part of *arg* as an integer. This function corresponds to `fn:day-from-dateTime`. More

Continued on next page

Table 3 – continued from previous page

Function	Description
`xsd:integer HOURS(xsd:dateTime arg)`	Returns the hours part of *arg* as an integer. The value is as given in the lexical form of the XSD dateTime. This function corresponds to `fn:hours-from-dateTime`. More
`xsd:integer MINUTES(xsd:dateTime arg)`	Returns the minutes part of the lexical form of *arg*. The value is as given in the lexical form of the XSD dateTime. This function corresponds to `fn:minutes-from-dateTime`. More
`xsd:decimal SECONDS(xsd:dateTime arg)`	Returns the seconds part of the lexical form of *arg*. This function corresponds to `fn:seconds-from-dateTime`. More
`xsd:dayTimeDuration TIMEZONE(xsd:dateTime arg)`	Returns the timezone part of *arg* as an xsd:dayTimeDuration. Raises an error if there is no timezone. This function corresponds to `fn:timezone-from-dateTime` except for the treatment of literals with no timezone. More
`xsd:string TZ(xsd:dateTime arg)`	Returns the timezone part of `arg` as a simple literal. Returns the empty string if there is no timezone. More
`xsd:string MD5(xsd:string arg)`	Returns the MD5 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. More
`xsd:string SHA1(xsd:string arg)`	Returns the SHA1 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. More
`xsd:string SHA256(xsd:string arg)`	Returns the SHA256 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. More
`xsd:string SHA512(xsd:string arg)`	Returns the SHA512 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. More

8.6.2 SPARQL 1.1 constructor functions

Casting in SPARQL 1.1 is performed by calling a constructor function for the target type on an operand of the source type. The standard includes the following constructor functions:

Note: Note that SPARQL 1.1 **does not** have an `xsd:date` constructor. Instead, use `STRDT(value,xsd:date)` to attach the `xsd:date` datatype to the value.

Constructor function	Description
`literal STRDT(xsd:string lexicalForm, iri datatypeIRI)`	The STRDT function constructs a literal with lexical form and type as specified by the arguments. More
`xsd:langString STRLANG(xsd:string lexicalForm, xsd:string langTag)`	The STRLANG function constructs a literal with lexical form and language tag as specified by the arguments. More
`xsd:integer(rdfTerm value)`	Casts *value* to `xsd:integer`. More
`xsd:decimal(rdfTerm value)`	Casts *value* to `xsd:decimal`. More
`xsd:float(rdfTerm value)`	Casts *value* to `xsd:float`. More
`xsd:double(rdfTerm value)`	Casts *value* to `xsd:double`. More
`xsd:string(rdfTerm value)`	Casts *value* to `xsd:string`. More
`xsd:boolean(rdfTerm value)`	Casts *value* to `xsd:boolean`. More
`xsd:dateTime(rdfTerm value)`	Casts *value* to `xsd:dateTime`. More
`xsd:nonPositiveInteger(rdfTerm value)`	Casts *value* to `xsd:nonPositiveInteger`. More
`xsd:negativeInteger(rdfTerm value)`	Casts *value* to `xsd:negativeInteger`. More
`xsd:long(rdfTerm value)`	Casts *value* to `xsd:long`. More
`xsd:int(rdfTerm value)`	Casts *value* to `xsd:int`. More
`xsd:short(rdfTerm value)`	Casts *value* to `xsd:short`. More
`xsd:byte(rdfTerm value)`	Casts *value* to `xsd:byte`. More
`xsd:nonNegativeInteger(rdfTerm value)`	Casts *value* to `xsd:nonNegativeInteger`. More
`xsd:unsignedLong(rdfTerm value)`	Casts *value* to `xsd:unsignedLong`. More
`xsd:unsignedInt(rdfTerm value)`	Casts *value* to `xsd:unsignedInt`. More
`xsd:unsignedShort(rdfTerm value)`	Casts *value* to `xsd:unsignedShort`. More
`xsd:unsignedByte(rdfTerm value)`	Casts *value* to `xsd:unsignedByte`. More
`xsd:positiveInteger(rdfTerm value)`	Casts *value* to `xsd:positiveInteger`. More

8.7 SPARQL extensions reference

This page describes functions that are not part of the W3C SPARQL specification but that are available for use to use in GraphDB.

8.7.1 Mathematical function extensions

Beside the standard SPARQL functions operating on numbers, GraphDB offers several additional functions, allowing users to do more mathematical operations. These are implemented using Java's Math class.

The prefix `ofn:` stands for the namespace `<http://www.ontotext.com/sparql/functions/>`.

Function	Description
`xsd:double ofn:acos(numeric a)`	The arccosine function. The input is in the range $[-1, +1]$. The output is in the range $[0, \pi]$ radians. See Math.acos(double). Example: `ofn:acos(0.5) = 1.0471975511965979`
`xsd:double ofn:asin(numeric a)`	The arcsine function. The input is in the range $[-1, +1]$. The output is in the range $[-\pi/2, \pi/2]$ radians. See Math.asin(double). Example: `ofn:asin(0.5) = 0.5235987755982989`
`xsd:double ofn:atan(numeric a)`	The arctangent function. The output is in the range $(-\pi/2, \pi/2)$ radians. See Math.atan(double). Example: `ofn:atan(1) = 0.7853981633974483`
`xsd:double ofn:atan2(numeric y, numeric x)`	The double-argument arctangent function (the angle component of the conversion from rectangular coordinates to polar coordinates). The output is in the range $[-\pi/2, \pi/2]$ radians. See Math.atan2(double,double). Example: `ofn:atan2(1, 0) = 1.5707963267948966`
`xsd:double ofn:cbrt(numeric a)`	The cubic root function. See Math.cbrt(double). Example: `ofn:cbrt(2) = 1.2599210498948732`
`xsd:double ofn:copySign(numeric magnitude, numeric sign)`	Returns the first floating-point argument with the sign of the second floating-point argument. See Math.copySign(double,double). Example: `ofn:copySign(2, -7.5) = -2.0`
`xsd:double ofn:cos(numeric a)`	The cosine function. The argument is in radians. See Math.cos(double). Example: `ofn:cos(1) = 0.5403023058681398`
`xsd:double ofn:cosh(numeric x).`	The hyperbolic cosine function. See Math.cosh(double). Example: `ofn:cosh(1) = 1.543080634815244`
`xsd:double ofn:e()`	Returns the double value that is closer than any other to e, the base of the natural logarithms. See Math.E. Example: `ofn:e() = 2.718281828459045`
`xsd:double ofn:exp(double a)`	The exponent function, e^x. See Math.exp(double). Example: `ofn:exp(2) = 7.38905609893065`
`xsd:double ofn:expm1(numeric x)`	Returns $e^x - 1$. See Math.expm1(double). Example: `ofn:expm1(3) = 19.085536923187668`
`xsd:double ofn:floorDiv(numeric x, numeric y)`	Returns the largest (closest to positive infinity) int value (as a double number) that is less than or equal to the algebraic quotient. The arguments are implicitly cast to long. See Math.floorDiv(long,long). Example: `ofn:floorDiv(5, 2) = 2.0`
`xsd:double ofn:floorMod(numeric x, numeric y)`	Returns the floor modulus (as a double number) of the arguments. The arguments are implicitly cast to long. See Math.floorMod(long,long). Example: `ofn:floorMod(10, 3) = 1.0`
`xsd:double ofn:getExponent(numeric d)`	Returns the unbiased exponent used in the representation of a double. This means that we take *n* from the binary representation of *x*: $x = 1 \times 2^n + \{1\|0\} \times 2^{(n-1)} + ... + \{1\|0\} \times 2^0$, i.e., the power of the highest non-zero bit of the binary form of *x*. See Math.getExponent(double). Example: `ofn:getExponent(10) = 3.0`

Continued on next page

Table 4 – continued from previous page

Function	Description
`xsd:double ofn:hypot(numeric x, numeric y)`	Returns $sqrt(x^2 + y^2)$ without intermediate overflow or underflow. See Math.hypot(double,double). Example: `ofn:hypot(3, 4) = 5.0`
`xsd:double ofn:IEEEremainder(numeric f1, numeric f2)`	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard. See Math.IEEEremainder(double,double). Example: `ofn:IEEEremainder(3, 4) = -1.0`
`xsd:double ofn:log(numeric a)`	The natural logarithm function. See Math.log(double). Example: `ofn:log(4) = 1.3862943611198906`
`xsd:double ofn:log10(numeric a).`	The common (decimal) logarithm function. See Math.log10(double). Example: `ofn:log10(4) = 0.6020599913279624`
`xsd:double ofn:log1p(numeric x)`	Returns the natural logarithm of the sum of the argument and 1. See Math.log1p(double). Example: `ofn:log1p(4) = 1.6094379124341003`
`xsd:double ofn:max(numeric a, numeric b)`	The greater of two numbers. See Math.max(double,double). Example: `ofn:max(3, 5) = 5.0`
`xsd:double ofn:min(numeric a, numeric b)`	The smaller of two numbers. See Math.min(double,double). Example: `ofn:min(3, 5) = 3.0`
`xsd:double ofn:nextAfter(numeric start, numeric direction)`	Returns the floating-point number adjacent to the first argument in the direction of the second argument. See Math.nextAfter(double,double). Example: `ofn:nextAfter(2, -7) = 1.9999999999999998`
`xsd:double ofn:nextDown(numeric d)`	Returns the floating-point value adjacent to *d* in the direction of negative infinity. See Math.nextDown(double). Example: `ofn:nextDown(2) = 1.9999999999999998`
`xsd:double ofn:nextUp(numeric d)`	Returns the floating-point value adjacent to *d* in the direction of positive infinity. See Math.nextUp(double). Example: `ofn:nextUp(2) = 2.0000000000000004`
`xsd:double ofn:pi()`	Returns the double value that is closer than any other to π, the ratio of the circumference of a circle to its diameter. See Math.PI. Example: `ofn:pi() = 3.141592653589793`
`xsd:double ofn:pow(numeric a, numeric b)`	The power function. See Math.pow(double,double). Example: `ofn:pow(2, 3) = 8.0`
`xsd:double ofn:rint(numeric a)`	Returns the double value that is closest in value to the argument and is equal to a mathematical integer. See Math.rint(double). Example: `ofn:rint(2.51) = 3.0`
`xsd:double ofn:scalb(numeric d, numeric scaleFactor)`	Returns $d \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set. See Math.scalb(double,int). `scaleFactor` can be negative, for example: `ofn:scalb(3, -3) = 3 * 2^-3 = 0.375`. Example: `ofn:scalb(3, 3) = 24.0`

Continued on next page

Table 4 – continued from previous page

Function	Description
`xsd:double ofn:signum(numeric d)`	Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero. See Math.signum(double). Example: `ofn:signum(-5) = -1.0`
`xsd:double ofn:sin(numeric a)`	The sine function. The argument is in radians. See Math.sin(double). Example: `ofn:sin(2) = 0.9092974268256817`
`xsd:double ofn:sinh(numeric x)`	The hyperbolic sine function. See Math.sinh(double). Example: `ofn:sinh(2) = 3.626860407847019`
`xsd:double ofn:sqrt(numeric a)`	The square root function. See Math.sqrt(double). Example: `ofn:sqrt(2) = 1.4142135623730951d`
`xsd:double ofn:tan(numeric a)`	The tangent function. The argument is in radians. See Math.tan(double). Example: `ofn:tan(1) = 1.5574077246549023`
`xsd:double ofn:tanh(numeric x)`	The hyperbolic tangent function. See Math.tanh(double). Example: `ofn:tanh(1) = 0.7615941559557649`
`xsd:double ofn:toDegrees(numeric angrad)`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees. See Math.toDegrees(double). Example: `ofn:toDegrees(1) = 57.29577951308232`
`xsd:double ofn:toRadians(numeric angdeg)`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians. See Math.toRadians(double). Example: `ofn:toRadians(1) = 0.017453292519943295`
`xsd:double ofn:ulp(numeric d)`	Returns the size of an ulp of the argument. An ulp, unit in the last place, of a double value is the positive distance between this floating-point value and the double value next larger in magnitude. Note that for non-NaN x, ulp(-x) == ulp(x). See Math.ulp(double). Example: `ofn:ulp(1) = 2.220446049250313E-16`

GraphDB also supports several Jena ARQ simple mathematical function analogs. The prefix `afn:` stands for the namespace `<http://jena.apache.org/ARQ/function#>`. (See *Jena SPARQL extensions* for additional Jena functions supported by GraphDB.)

Function	Description
`afn:min(num1, num2)`	Return the minimum of two numbers.
`afn:max(num1, num2)`	Return the maximum of two numbers.
`afn:pi()`	The value of pi as an XSD double.
`afn:e()`	The value of e as an XSD double.
`afn:sqrt(num)`	The square root of num.

8.7.2 Date and time function extensions

Beside the standard SPARQL functions related to date and time, GraphDB offers several additional functions, allowing users to do more with their temporal data.

The prefix `ofn:` stands for the namespace `<http://www.ontotext.com/sparql/functions/>`. For more information, refer to *Time functions extensions*.

Function	Description
`xsd:long ofn:years-from-duration(xsd:duration dur)`	Return the “years” part of the duration literal
`xsd:long ofn:months-from-duration(xsd:duration dur)`	Returns the “months” part of the duration literal
`xsd:long ofn:days-from-duration(xsd:duration dur)`	Returns the “days” part of the duration literal
`xsd:long ofn:hours-from-duration(xsd:duration dur)`	Returns the “hours” part of the duration literal
`xsd:long ofn:minutes-from-duration(xsd:duration dur)`	Returns the “minutes” part of the duration literal
`xsd:long ofn:seconds-from-duration(xsd:duration dur)`	Returns the “seconds” part of the duration literal
`xsd:long ofn:millis-from-duration(xsd:duration dur)`	Returns the “milliseconds” part of the duration literal
`xsd:long ofn:asWeeks(xsd:duration dur)`	Returns the duration of the period as weeks
`xsd:long ofn:asDays(xsd:duration dur)`	Returns the duration of the period as days
`xsd:long ofn:asHours(xsd:duration dur)`	Returns the duration of the period as hours
`xsd:long ofn:asMinutes(xsd:duration dur)`	Returns the duration of the period as minutes
`xsd:long ofn:asSeconds(xsd:duration dur)`	Returns the duration of the period as seconds
`xsd:long ofn:asMillis(xsd:duration dur)`	Returns the duration of the period as milliseconds
`xsd:long ofn:weeksBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as weeks
`xsd:long ofn:daysBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as days
`xsd:long ofn:hoursBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as hours
`xsd:long ofn:minutesBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as minutes
`xsd:long ofn:secondsBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as seconds
`xsd:long ofn:millisBetween(xsd:dateTime d1, xsd:dateTime d2)`	Returns the duration between the two dates as milliseconds

8.7.3 RDF-star extension functions

To avoid any parsing of an embedded triple, GraphDB introduces the following SPARQL functions:

Function	Description
`xsd:boolean rdf:isTriple(variable var)`	Checks if the variable *var* is bound to an embedded triple

`iri rdf:subject(variable var)`
`iri rdf:predicate(variable var)`
`rdfTerm rdf:object(variable var)`
	Extracts the subject, predicate, or object from a variable bound to an embedded triple

`rdf:Statement(iri subj, iri pred, rdfTerm obj)`	Creates a new embedded statement with the provided values

See more about RDF-star and SPARQL-star syntax in the *Syntax and examples* section of this documentation's *RDF-star and SPARQL-star* page.

8.7.4 SPARQL functions vs magic predicates

Functions and magic predicates are denoted and used differently. Magic predicates are similar to how GraphDB plugins can interpret certain triple patterns, and unlike functions, they can return multiple values per call.

- Function calls take the form `ex:function(arg1, arg2, ...)`, where all arguments must be bound. These are most commonly used in `BIND`, `SELECT`, `FILTER`, and `ORDER BY` clauses but may be used in other places as well.
- Magic predicates take the form `subject ex:magicPredicate (arg1 arg2 ...)`. In some cases, the arguments are allowed to be unbound, and are then calculated from the `subject`. They are used as query triple patterns that have the magic predicate name in the middle, a parenthesized RDF list of arguments as the object, and the variable where the result should be bound (or a parenthesized list of such variables, in the case of certain *LLM GPT extensions*) as the subject.

8.7.5 SPARQL SPIN functions

The following SPIN SPARQL functions and magic predicates are available in GraphDB. The prefix `spif:` stands for the namespace `<http://spinrdf.org/spif#>`.

SPIN functions that work on text use 0-based indexes, unlike SPARQL's functions, which use 1-based indexes.

	Function	Description
Dates	`spif:parseDate(xsd:string date, xsd:string format)`	Parses *date* using *format* with Java's SimpleDateFormat
	`spif:dateFormat(xsd:dateTime date, xsd:string format)`	Formats *date* using *format* with Java SimpleDateFormat
	`xsd:long spif:currentTimeMillis()`	The current time in milliseconds since the epoch
	`xsd:long spif:timeMillis(xsd:dateTime date)`	The time in milliseconds since the epoch for the provided argument
Numbers	`numeric spif:mod(xsd:numeric dividend, xsd:numeric divisor)`	Remainder from integer division
	`xsd:string spif:decimalFormat(numeric number, xsd:string format)`	Formats *number* using *format* with Java's DecimalFormat
	`xsd:double spif:random()`	Calls Java's Math.random()
Strings	`xsd:string spif:trim(string str)`	Calls `String.trim()`
	`spif:generateUUID()`	UUID generation as a literal. Same as SPARQL's `STRUUID()`.
	`spif:cast(literal value, iri type)`	Same as SPARQL's `STRDT(STR(value), type)`. Does not do validation either.
	`xsd:int spif:indexOf(string str, string substr)`	Position of first occurrence of a substring. Note that SPIN functions that work on text use 0-based indexes, unlike SPARQL's functions, which use 1-based indexes.
	`xsd:int spif:lastIndexOf(string str, string substr)`	Position of last occurrence of a substring. Note that SPIN functions that work on text use 0-based indexes, unlike SPARQL's functions, which use 1-based indexes.
	`xsd:string spif:buildString(string template, arguments…)`	Builds a literal from a template, e.g. "foo {?2} bar {?1}" where `{?2}` will be replaced with second argument and `{?1}` will be replaced with the first argument after the template.
	`xsd:string spif:replaceAll(string str, xsd:string regexp, xsd:string flags)`	Calls Java `String.replaceAll`, same as SPARQL's `REPLACE()`.
	`xsd:string spif:unCamelCase(string str)`	Converts camel-cased string to non-camel case string with spaces
	`xsd:string spif:upperCase(string str)`	Converts to upper case, similar to SPARQL's `UCASE()` but disregarding the language tag from the input string
	`xsd:string spif:lowerCase(string str)`	Converts to lower case, similar to SPARQL's `LCASE()` but disregarding the language tag from the input

There are three magic predicates: `spif:split`, `spif:for`, and `spif:foreach`.

Predicate	Description

`?result spif:split (?string ?regex)`	Takes two arguments: a *string* to split and a *regex* to split on. The current implementation uses Java's `String.split()` method.
	The following query splits the `?input` string at the commas and returns three rows, with the `?name` value being "Marion", then "Suess", then "Green" in those rows:
	`PREFIX spif: <http://spinrdf.org/spif#>`
	
	`SELECT ?name WHERE {`
	`  BIND(str("Marion,Suess,Green") as ?input)`
	`  ?name spif:split (?input ",") .`
	`}`

`?result spif:for (?start ?end)`	Generates bindings from a given *start* integer value to another given *end* integer value.
	The following query returns three rows, with the variable `?result` having the value 6 in the first row, 7 in the second, and 8 in the third:
	`PREFIX spif: <http://spinrdf.org/spif#>`
	
	`SELECT ?result WHERE {`
	`  ?result spif:for (6 8)`
	`}`

`?result spif:foreach (?arg1 ?arg2 …)`	Generates bindings for the given arguments *arg1*, *arg2* and so on. They can be of any type.
	The following query returns three rows, with each row showing `?result` bound to one of the arguments being passed:
	`PREFIX spif: <http://spinrdf.org/spif#>`
	`PREFIX ex:  <http://www.example.com/ns/>`
	
	`SELECT ?result WHERE {`
	`  ?result spif:foreach (ex:some-uri "some string" 23.7)`
	`}`

8.7.6 RDF list function extensions

GraphDB supports the below Jena list function analogs.

The prefix `list:` stands for the namespace `<http://jena.apache.org/ARQ/list#>`.

Function	Description
`list list:member member`	Membership of an RDF List (RDF Collection). Currently in GraphDB, if `list` is not bound or a constant, an error will be thrown; else evaluate for the list in the variable `list`. If `member` is a variable, generate solutions with `member` bound to each element of list. If `member` is bound or a constant expression, test to see if it is a member of `list`.
`list list:index (index member)`	Index of an RDF List (RDF Collection). Currently in GraphDB, if `list` is not bound or a constant, an error will be thrown; else evaluate for one particular list. The object is a list pair, either element can be bound, unbound or a fixed node. Unbound variables in the object list are bound by the property function.
`list list:length length`	Length of an RDF List (RDF Collection). Currently in GraphDB, if `list` is not bound or a constant, an error will be thrown; else evaluate for one particular list. The object is tested against or bound to the length of the list.

Note: The Jena behavior is that if `list` is not bound or a constant, the function finds and iterates all lists in the graph (can be slow). As mentioned above, currently, GraphDB does not provide support for unbound `list`. Support for it will be added with the coming releases.

8.7.7 Aggregation function extensions

GraphDB supports the below Jena ARQ aggregate function analogs, which are modeled after the corresponding SQL aggregate functions.

The prefix `agg:` stands for the namespace `<http://jena.apache.org/ARQ/function/aggregate#>`.

- `agg:stdev`
- `agg:stdev_samp`
- `agg:stdev_pop`
- `agg:variance`
- `agg:var_samp`
- `agg:var_pop`

The `stdev_pop()` and `stdev_samp()` functions compute the population standard deviation and sample standard deviation, respectively, of the input values. (`stdev()` is an alias for `stdev_samp()`) Both functions evaluate all input rows matched by the query. The difference is that `stdev_samp()` is scaled by 1/(N-1) while `stdev_pop()` is scaled by 1/N.

The `var_samp()` and `var_pop()` functions compute the sample variance and population variance, respectively, of the input values. (`variance()` is an alias for `variance_samp()`) Both functions evaluate all input rows matched by the query. The difference is that `variance_samp()` is scaled by 1/(N-1) while `variance_pop()` is scaled by 1/N.

8.7.8 GeoSPARQL functions

The following functions are defined by the GeoSPARQL standard. For more information, refer to OGC GeoSPARQL - A Geographic Query Language for RDF Data. The prefix `geof:` stands for the namespace `<http://www.opengis.net/def/function/geosparql/>`.

The type `geomLiteral` serves as a placeholder for any GeoSPARQL literal that describes a geometry. GraphDB supports WKT (datatype `geo:wktLiteral`) and GML (datatype `geo:gmlLiteral`).

Function	Description
`xsd:double geof:distance(geomLiteral geom1, geomLiteral geom2, iri units)`	Returns the shortest distance in units between any two Points in the two geometric objects is calculated in the spatial reference system of *geom1*. See the list following this table for potential unit values to pass as the `iri` parameter.
`geomLiteral geof:buffer(geomLiteral geom, xsd:double radius, iri units)`	This function returns a geometric object that represents all Points whose distance from *geom1* is less than or equal to the radius measured in units. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:convexHull(geomLiteral geom1)`	This function returns a geometric object that represents all Points in the convex hull of *geom1*. Calculations are in the spatial reference system of *geom1*.

Continued on next page

Table 5 – continued from previous page

Function	Description
`geomLiteral geof:intersection(geomLiteral geom1, geomLiteral geom2)`	This function returns a geometric object that represents all Points in the intersection of *geom1* with *geom2*. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:union(geomLiteral geom1, geomLiteral geom2)`	This function returns a geometric object that represents all Points in the union of *geom1* with *geom2*. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:difference(geomLiteral geom1, geomLiteral geom2)`	This function returns a geometric object that represents all Points in the set difference of *geom1* with *geom2*. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:symDifference(geomLiteral geom1, geomLiteral geom2)`	This function returns a geometric object that represents all Points in the set symmetric difference of *geom1* with *geom2*. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:envelope(geomLiteral geom1)`	This function returns the minimum bounding box of *geom1*. Calculations are in the spatial reference system of *geom1*.
`geomLiteral geof:boundary(geomLiteral geom1)`	This function returns the closure of the boundary of *geom1*. Calculations are in the spatial reference system of *geom1*.
`iri geof:getSRID(geomLiteral geom)`	Returns the spatial reference system URI for *geom*.
`xsd:boolean geof:relate(geomLiteral geom1, geomLiteral geom2, xsd:string matrix)`	Returns true if the spatial relationship between *geom1* and *geom2* corresponds to one of acceptable values for the specified pattern-matrix. Otherwise, this function returns false. Pattern-matrix represents a DE-9IM intersection pattern consisting of T (true) and F (false) values. The spatial reference system for *geom1* is used for spatial calculations.
`xsd:boolean geof:sfEquals(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFFFTFFFT)`
`xsd:boolean geof:sfDisjoint(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FF*FF****)`
`xsd:boolean geof:sfIntersects(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T******** *T******* ***T***** ****T****)`
`xsd:boolean geof:sfTouches(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FT******* F**T***** F***T****)`
`xsd:boolean geof:sfCrosses(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*T***T**)` for P/L, P/A, L/A; `(0*T***T**)` for L/L
`xsd:boolean geof:sfWithin(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*F**F***)`

Continued on next page

Table 5 – continued from previous page

Function	Description
`xsd:boolean geof:sfContains(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*****FF*)`
`xsd:boolean geof:sfOverlaps(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*T***T**)` for A/A, P/P; `(1*T***T**)` for L/L
`xsd:boolean geof:ehEquals(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFFFTFFFT)`
`xsd:boolean geof:ehDisjoint(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FF*FF****)`
`xsd:boolean geof:ehMeet(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FT******* F**T***** F***T****)`
`xsd:boolean geof:ehOverlap(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*T***T**)`
`xsd:boolean geof:ehCovers(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*TFT*FF*)`
`xsd:boolean geof:ehCoveredBy(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFF*TFT**)`
`xsd:boolean geof:ehInside(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFF*FFT**)`
`xsd:boolean geof:ehContains(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(T*TFF*FF*)`
`xsd:boolean geof:rcc8eq(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFFFTFFFT)`
`xsd:boolean geof:rcc8dc(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FFTFFTTTT)`
`xsd:boolean geof:rcc8ec(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(FFTFTTTTT)`

Continued on next page

Table 5 – continued from previous page

Function	Description
`xsd:boolean geof:rcc8po(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TTTTTTTTT)`
`xsd:boolean geof:rcc8tppi(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TTTFTTFFT)`
`xsd:Boolean geof:rcc8tpp(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFFTTFTTT)`
`xsd:boolean geof:rcc8ntpp(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TFFTFFTTT)`
`xsd:boolean geof:rcc8ntppi(geomLiteral geom1, geomLiteral geom2)`	DE-9IM intersection pattern: `(TTTFFTFFT)`

The following are potential `geof:distance` unit values:

- http://www.opengis.net/def/uom/OGC/1.0/degree
- http://www.opengis.net/def/uom/OGC/1.0/GridSpacing
- http://www.opengis.net/def/uom/OGC/1.0/metre
- http://www.opengis.net/def/uom/OGC/1.0/radian
- http://www.opengis.net/def/uom/OGC/1.0/unity

8.7.8.1 GeoSPARQL extension functions

On top of the standard GeoSPARQL functions, GraphDB adds a few useful extensions based on the USeekM library. The prefix `geoext:` stands for the namespace `<http://rdf.useekm.com/ext#>`.

The types `geo:Geometry`, `geo:Point`, etc. refer to GeoSPARQL types in the `http://www.opengis.net/ont/geosparql#` namespace.

See more about GraphDB's GeoSPARQL extensions in the *GeoSPARQL support* section of this documentation.

Function	Description
`xsd:double geoext:area(geomLiteral g)`	Calculates the area of the surface of the geometry.
`geomLiteral geoext:closestPoint(geomLiteral g1, geomLiteral g2)`	For two given geometries, computes the point on the first geometry that is closest to the second geometry.
`xsd:boolean geoext:containsProperly(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry properly contains the second geometry. *Geom1* contains properly *geom2* if all *geom1* contains *geom2* and the boundaries of the two geometries *do not* intersect.
`xsd:boolean geoext:coveredBy(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry is covered by the second geometry. *Geom1* is covered by *geom2* if every point of *geom1* is a point of *geom2*.
`xsd:boolean geoext:covers(geomLiteral g1, geomLiteral g2)`	Tests if the first geometry covers the second geometry. *Geom1* covers *geom2* if every point of *geom2* is a point of *geom1*.
`xsd:double geoext:hausdorffDistance(geomLiteral g1, geomLiteral g2)`	Measures the degree of similarity between two geometries. The measure is normalized to lie in the range [0, 1]. Higher measures indicate a greater degree of similarity.
`geo:Line geoext:shortestLine(geomLiteral g1, geomLiteral g2)`	Computes the shortest line between two geometries. Returns it as a LineString object.
`geomLiteral geoext:simplify(geomLiteral g, double d)`	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm.
`geomLiteral geoext:simplifyPreserveTopology(geomLiteral g, double d)`	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm. Will avoid creating derived geometries (polygons in particular) that are invalid.
`xsd:boolean geoext:isValid(geomLiteral g)`	Checks whether the input geometry is a valid geometry.

8.7.9 Geospatial extension functions

At present, there is just one SPARQL extension function. The prefix `omgeo:` stands for the namespace `<http://www.ontotext.com/owlim/geo#>`.

Function	Description
`xsd:double omgeo:distance(numeric lat1, numeric long1, numeric lat2, numeric long2)`	Computes the distance between two points in kilometers and can be used in `FILTER` and `ORDER BY` clauses. Latitude is limited to the range -90 (South) to +90 (North). Longitude is limited to the range -180 (West) to +180 (East).

See more about *GraphDB's geospatial extension functions*.

8.7.10 Jena SPARQL extensions

Below are several Jena function analogs supported by GraphDB. (See *Mathematical function extensions* for additional Jena math functions that GraphDB supports.)

The prefix `afn:` stands for the namespace `<http://jena.apache.org/ARQ/function#>`.

Function	Description
`iri apf:splitIRI (namespace localname)` `iri apf:splitURI (namespace localname)`	Split the IRI or URI into namespace (an IRI) and local name (a string). Compare if given values or bound variables, otherwise set the variable. The object is a list with 2 elements. `splitURI` is a synonym.
`var apf:concat (arg arg …)`	Concatenate the arguments in the object list as strings, and assign to `var`.
`var apf:strSplit (arg arg)`	Split a string and return a binding for each result. The subject variable should be unbound. The first argument to the object list is the string to be split. The second argument to the object list is a regular expression by which to split the string. The subject `var` is bound for each result of the split, and each result has the whitespace trimmed from it.
`afn:bnode(?x)`	Return the blank node label if `?x` is a blank node.
`afn:localname(?x)`	The local name of `?x`.
`afn:namespace(?x)`	The namespace of `?x`.
`afn:sprintf(format, v1, v2, …)`	Make a string from the format string and the RDF terms.
`afn:substr(string, startIndex [,endIndex])`	Substring, Java style using `startIndex` and `endIndex`.
`afn:substring`	Synonym for `afn:substr`.
`afn:strjoin(sep, string …)`	Concatenate given strings, using `sep` as a separator.
`afn:sha1sum(resource)`	Calculate the SHA1 checksum of a literal or URI.
`afn:now()`	Current time. (Actually, a fixed moment of the current query execution — see the *standard function* `NOW()` for details.)

8.7.11 Time functions extensions

Beside the standard SPARQL functions related to time, GraphDB offers several additional functions, allowing users to do more with their time data. Those are implemented within the same namespace as standard math functions, `<http://www.ontotext.com/sparql/functions/>`. The default prefix for the functions is `ofn`.

8.7.11.1 Period extraction functions

The first group of functions is related to accessing particular parts of standard duration literals. For example, the expression `2019-03-24T22:12:29.183+02:00" - "2019-04-19T02:42:28.182+02:00"` will produce the following duration literal: `-P25DT4H29M58.999S`. It is possible to parse the result and obtain the proper parts of it - for example, “25 days”, “4” hours, or more discrete time units. However, instead of having to do this manually, GraphDB offers functions that perform the computations at the engine level. The functions take a period as input and output `xsd:long`.

Note: The functions described here perform simple extractions, rather than computing the periods. For example, if you have 40 days in the duration literal, but no months, i.e., `P0Y0M40DT4H29M58.999S`, a `months-from-duration` extraction will not return 1 months.

The following table describes the functions that are implemented and gives example results, assuming the literal `-P25DT4H29M58.999S` is passed to them:

Function | Description		Expected return value
`ofn:years-from-duration` | Return the "years" part of the duration literal		0
`ofn:months-from-duration`	Returns the "months" part of the duration literal	0
`ofn:days-from-duration`	Returns the "days" part of the duration literal	25
`ofn:hours-from-duration`	Returns the "hours" part of the duration literal	4
`ofn:minutes-from-duration`	Returns the "minutes" part of the duration literal	29
`ofn:seconds-from-duration`	Returns the "seconds" part of the duration literal	58
`ofn:millis-from-duration`	Returns the "milliseconds" part of the duration literal	999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
    bind (ofn:millis-from-duration("-P25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```

8.7.11.2 Period transformation functions

The second group of functions is related to transforming a standard duration literal. This reduces the need for performing mathematical transformations on the input date. The functions take a period as input and output `xsd:long`.

Note: The transformation is performed with no fractional components. For example, if transformed, the duration literal we used previously, `-P25DT4H29M58.999S` will yield 25 days, rather than 25.19 days.

The following table describes the functions that are implemented and gives example results, assuming the literal `-P25DT4H29M58.999S` is passed to them. Note that the return values are negative since the period is negative:

Function	Description	Expected return value
`ofn:asWeeks`	Returns the duration of the period as weeks	-3
`ofn:asDays`	Returns the duration of the period as days	-25
`ofn:asHours`	Returns the duration of the period as hours	-604
`ofn:asMinutes`	Returns the duration of the period as minutes	-36269
`ofn:asSeconds`	Returns the duration of the period as seconds	-2176198
`ofn:asMillis`	Returns the duration of the period as milliseconds	-2176198999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
    bind (ofn:asMillis("-P25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```

8.7.11.3 Durations expressed in certain units

The third group of functions eliminates the need for computing a difference between two dates when a transformation will be necessary, essentially combining the mathematical operation of subtracting two dates with a transformation. It is more efficient than performing an explicit mathematical operation between two date literals, for example: `"2019-03-24T22:12:29.183+02:00" - "2019-04-19T02:42:28.182+02:00"` and then using a transformation function. The functions take two dates as input and output integer literals.

Note: Regular SPARQL subtraction can return negative values, as evidenced by the negative duration literal used in the example. However, comparisons are only positive. So, comparison isn't an exact match for a subtraction followed by transformation. If one of the timestamps has timezone but the other does not, the result is ill-defined.

The following table describes the functions that are implemented and gives example results, assuming the date literals `2019-03-24T22:12:29.183+02:00"` and `"2019-04-19T02:42:28.182+02:00"` are passed to them. Note that the return values are positive:

Function	Description	Expected return value
`ofn:weeksBetween`	Returns the duration between the two dates as weeks	3
`ofn:daysBetween`	Returns the duration between the two dates as days	25
`ofn:hoursBetween`	Returns the duration between the two dates as hours	604
`ofn:minutesBetween`	Returns the duration between the two dates as minutes	36269
`ofn:secondsBetween`	Returns the duration between the two dates as seconds	2176198
`ofn:millisBetween`	Returns the duration between the two dates as milliseconds	2176198999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
    bind (ofn:millisBetween("2019-03-24T22:12:29.183+02:00"^^xsd:dateTime, "2019-04-19T02:42:28.
↪182+02:00"^^xsd:dateTime) as ?result)
}
```

8.7.11.4 Arithmetic operations

The fourth group of functions includes operations such as: adding duration to a date; adding dayTimeDuration to a dateTime; adding time duration to a time; comparing durations. This is done via the SPARQL operator extensibility.

8.7.12 LLM GPT extensions

These magic predicates send queries to an LLM GPT model and return the answer as part of your SPARQL query results. Before using any of them, you must configure your access to the model as described in *Talk to Your Graph properties* and *SPARQL functions and SPARQL explain*.

See *GPT functions* for more information on these magic predicates.

Function	Description
`?answer gpt:ask ?message` `?answer gpt:ask (?message1 ?message2 …)`	This magic predicate passes one or more messages with instructions to the LLM GPT. The results (unlike with *gpt:list*) are stored in a single binding. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response. If passing more than one argument in the triple pattern's object position, enclose them in parentheses. See *gpt:ask() — Retrieve a single answer* for more information on this magic predicate.
`?answer gpt:list ?message` `?answer gpt:list (?message1 ?message2 …)`	This magic predicate passes one or more messages with instructions to the LLM GPT. These instructions should include a number that indicates how many results that you would like. The results (unlike with *gpt:ask*) can then be returned as multiple bindings of the specified variable, which will be displayed as separate result set rows. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response. If passing more than one argument in the triple pattern's object position, enclose them in parentheses. See *gpt:list() — Retrieve a list of answers* for more information on this magic predicate.

`?column1 gpt:table ?message`
`?column1 gpt:table (?message1 ?message2 …)`
`(?column1 ?column2) gpt:table ?message`
`(?column1 ?column2 ?column3) gpt:table (?message1 ?message2 …)`
	This sends one or more requests to create multiple bindings with multiple values — in other words, a table. The last message passed can be a real number between 0 and 2 to set the *temperature* of the response.
	Because `gpt:table` is a magic predicate, you call it as a triple pattern with the variable(s) in which to store the response as the subject and the messages to pass as the object. The subject can list more than one variable to store the values of the table columns. If you have more than one subject variable, enclose the list in parentheses. Similarly, if you pass more than one message, enclose the list in parentheses.
	See *gpt:table() — Retrieve a table of answers* for more information on this magic predicate.

8.7.13 List manipulation extension functions

Function	Description
`helper:tuple()`	Combines all of its arguments into an internal list that you can reference using a blank node connected to the list members. You can use this as both a function and as a magic predicate. The *helper:iterate() function* lets you access individual members of the list. See *helper:tuple() — Combine values into a list* for more on this function.
`helper:tupleAggr()`	Similar to the *helper:tuple* function but only takes one argument. If the argument is bound to multiple values, it will aggregate those values into a list that can be accessed by a blank node connected to the list members. You can access individual members of the list using the *helper:iterate() function*. See *helper:tupleAggr() — Aggregate values into a list* for more on this function.
`helper:rdf()`	Takes a subject, predicate, and object and combines them into an internal triple connected by a blank node. You can provide the triple components directly or combine using the *helper:tuple* function. If you do use the `helper:tuple` function, you may also add a reference to a named graph to store the triple. See *helper:rdf() — Combine values into an RDF triple* for more on this function.
`helper:serializeRDF()`	Serializes RDF stored internally with a function such as *helper:rdf* so that you can see the contents of the triples. The default format is Turtle-star, but you can request a different one by naming the MIME type as an optional second argument. See *helper:serializeRDF() — Convert internal RDF to readable triples* for more on this function.
`?listMember helper:iterate ?tuple`	Iterates over the elements of an internal list created by the `helper:tuple`, `helper:tupleAggr`, or `helper:rdf` functions. See *helper:iterate() — Iterate through an internal list* for more on this magic predicate.

8.8 GraphDB APIs

The following sections cover GraphDB APIs, which let you take advantage of its features from other applications:

- *Using the RDF4J REST API*: How to use the REST API that is included with GraphDB as part of the RDF4J framework. This includes support for query and update SPARQL endpoints.
- *Using the GraphDB REST API*: How to use the API that provides access to Workbench functions such as cluster, role, and location management.
- *Using GraphDB with the RDF4J API*: How to take advantage of the RDF4J Java API for GraphDB application development.
- *GraphDB Plugin API*: How to use the Java API to develop plugins that let you extend GraphDB.
- *Using Maven artifacts*: How to use libraries stored in the GraphDB Maven repository such as the database and client runtime jar files.

8.8.1 Using the RDF4J REST API

8.8.1.1 What is the RDF4J REST API?

GraphDB's use of the RDF4J framework means that GraphDB supports the RDF4J REST API. This API lets you create and delete repositories as well as add data to and delete and retrieve data from them. (The separate *GraphDB REST API* provides HTTP access to system management features such as cluster, role, and location management.) This RDF4J REST API support also includes support for W3C standard SPARQL query and update endpoints; this is described further below.

Selecting *Help* ▸ *REST API* from GraphDB's Workbench takes you to a quick reference for both the GraphDB REST API and the RDF4J REST API. The RDF4J project's RDF4J REST API documentation provides more detailed information about the range of API calls that it supports.

8.8.1.2 Using the RDF4J REST API's SPARQL endpoints

Most modern programming languages either have built-in support or can incorporate a library that supports the GET, PUT, POST, and DELETE operations of an HTTP interface. To demonstrate the use of the RDF4J REST API's SPARQL endpoints, examples here use the curl utility that is included with the Linux and macOS operating systems. (Windows users can easily download curl.)

A SPARQL endpoint is identified by its URL. Each repository stored in GraphDB has its own query and update URLs. For a starwars repository stored in a GraphDB instance that is available at http://localhost:7200, that repository's query endpoint URL would be http://localhost:7200/repositories/starwars.

Note: All examples shown in this section assume that security is off for the sample starwars repository being queried and updated. If it was turned on, standard HTTP authentication mechanisms such as the `curl --user` parameter or authentication tokens (or your programming language's equivalent) could be used for access to this repository.

Sending queries to a SPARQL endpoint

You can pass a query to the endpoint as part of the URL used to call `curl`, but because many characters in a typical SPARQL query are not legal URL characters, the query should be URL-encoded when submitted this way. For example, after URL-encoding a query like this,

```
SELECT * WHERE {?s ?p ?o} LIMIT 3
```

the `curl` command that passes this query as part of the URL might look like this:

```
curl -d "query=SELECT%20*%20WHERE%20%7B%3Fs%20%3Fp%20%3Fo%7D%20LIMIT%203" \
    http://localhost:7200/repositories/starwars
```

If you execute that `curl` command at the operating system command line (on a machine where a GraphDB instance with a starwars repository is running at that address), `curl` will display the returned results like this:

```
s,p,o
https://swapi.co/vocabulary/Amphibian,http://www.w3.org/1999/02/22-rdf-syntax-ns#type,http://www.w3.org/
↪2000/01/rdf-schema#Class
https://swapi.co/vocabulary/Artificial,http://www.w3.org/1999/02/22-rdf-syntax-ns#type,http://www.w3.org/
↪2000/01/rdf-schema#Class
https://swapi.co/vocabulary/Character,http://www.w3.org/1999/02/22-rdf-syntax-ns#type,http://www.w3.org/
↪2000/01/rdf-schema#Class
```

Escaped URLs in a `curl` command line can become unwieldy for all but the shortest queries. It's easier to store the query in a text file and then tell `curl` to send that file's query to the SPARQL endpoint. If the original `SELECT`

query above with no URL encoding was stored in a file called `myquery1.rq`, the following `curl` command would have the same effect as the previous one:

```
curl --data-urlencode "query@myquery1.rq" \
   http://localhost:7200/repositories/starwars
```

The result of the queries above were in CSV format. Other choices can be specified to `curl` using an HTTP Accept header with a MIME type passed with the `-H` option. This is especially useful if your SPARQL query is a `CONSTRUCT` query asking for triples or quads because you can specify the serialization you want. For example, if this next query is stored in the file `myquery2.rq`,

```
CONSTRUCT {?s ?p ?o}
WHERE {?s ?p ?o}
LIMIT 3
```

then the following `curl` command line would tell the endpoint to return the triples using the Turtle serialization:

```
curl --data-urlencode "query@myquery2.rq"  -H "Accept: text/turtle" \
      http://localhost:7200/repositories/starwars
```

GraphDB's *RDF formats* documentation page describes other MIME types that you can use such as `application/n-triples` and `application/ld+json`.

Sending update requests to a SPARQL endpoint

Sending an update request to a repository's update endpoint is similar to sending a query to a query endpoint. Before we demonstrate this, let's use the endpoint querying technique shown in the previous section to query for the name and height of the starwars repository resource `https://swapi.co/resource/rodian/15`:

```
PREFIX voc: <https://swapi.co/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT * WHERE {
    <https://swapi.co/resource/rodian/15> rdfs:label ?name ;
           voc:height ?height .
}
```

A `curl` command almost identical to the `myquery1.rq` one in the previous section, but sending a `myquery3.rq` file with this new query, shows us that the resource's name is "Greedo" and has a height of 173 centimeters:

```
name,height
Greedo,173
```

The following SPARQL update request updates this height value by deleting the triple that currently stores the resource's height and inserting a new one with a height value of 170.5:

```
PREFIX voc: <https://swapi.co/vocabulary/>

DELETE { <https://swapi.co/resource/rodian/15> voc:height ?o .  }
INSERT { <https://swapi.co/resource/rodian/15> voc:height 170.5 . }
WHERE  { <https://swapi.co/resource/rodian/15> voc:height ?o .  }
```

With this update request stored in a file called `updategreedo.ru`, this next `curl` command sends it to the SPARQL update endpoint. Note that the update endpoint URL is similar to the query endpoint used in the previous section but includes `/statements` at the end:

```
curl -X POST --data-urlencode "update@updategreedo.ru" \
    http://localhost:7200/repositories/starwars/statements
```

Successfully running this command line with no errors results in no output, so no news is good news. To verify that the update request did its job, re-running the `curl` command with the `myquery3.rq` query shows that we have successfully updated Greedo's `voc:height` value:

```
name,height
Greedo,170.5
```

8.8.1.3 Graph Store HTTP Protocol

Through the RDF4J REST API, GraphDB also fully supports the Graph Store HTTP Protocol for direct and indirect graph names. The SPARQL 1.1 Graph Store HTTP Protocol has the most details, although further information can be found in the RDF4J Server REST API documentation.

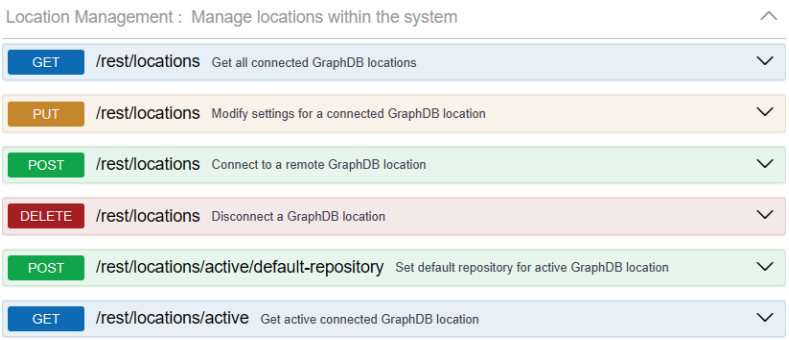
This protocol supports the management of RDF statements in named graphs in the REST style by providing the ability to get, delete, add to, or overwrite statement in named graphs using the basic HTTP methods.

8.8.2 Using the GraphDB REST API

The Workbench REST API can be used to automate various tasks without having to open the Workbench in a browser and doing them manually.

You can find more information about each REST API feature group and its operations under *Help* ▸ *REST API* in Workbench, as well as execute them directly from there and see the results.

On that REST API documentation, click on a feature group to expand it and see the operations it includes. Click an operation name to see details about it. Below, we see the Location Management operations.



The remainder of this document lists the major categories of REST API calls.

8.8.2.1 Access control list management

Use the access control list management API to manage your *Fine-grained access control* rules — view your access control lists (ACLs), add new rules to it or delete existing ones, or replace the ACL of a repository.

See also:

curl examples for managing FGAC

8.8.2.2 Authentication

Use the login REST API endpoint to obtain a GraphDB token in exchange for username and password.

See also:

curl examples for authentication

8.8.2.3 Cluster group management

Note: This feature requires a GraphDB Enterprise license.

Use the cluster group management API to create a cluster, view its configuration, monitor the status of both the cluster group and each of its nodes, as well as to delete the cluster.

See also:

curl examples for cluster group management

8.8.2.4 Custom roles management

Use the custom roles management API to view, grant or revoke custom user roles.

See also:

curl examples for managing custom user roles

8.8.2.5 Data import

Use the data import API to import data into GraphDB. You can import from a server or a remote URL.

See also:

curl examples for data import

8.8.2.6 Location management

Use the location management API to attach, edit, or detach GraphDB servers at remote locations.

See also:

curl examples for location management

8.8.2.7 Monitoring

Use the monitoring API to track running processes in GraphDB. The GraphDB REST API currently exposes five endpoints suitable for scraping by Prometheus — structures, repository, infrastructure, cluster and backup operations.

See also:

- The *Prometheus monitoring* section provides more information on the metrics that you can monitor as well as how to configure the Prometheus scrapers.
- curl examples for:
  - *structures statistics monitoring*
  - *repository monitoring*

- *infrastructure statistics monitoring*
- *cluster monitoring*
- *backup monitoring*

8.8.2.8 Recovery management

Use the recovery management API to create a backup or restore a GraphDB instance from a backup or from the cloud.

8.8.2.9 Repository management

Use the repository management API to add a repository to an attached location, remove it from one, or edit it. Unlike the RDF4J API, you can work with multiple remote locations from a single access point. When combined with the *Location management* API, you can use it to automate the creation of multiple repositories across your network.

See also:

curl examples for:

- *repository management*
- *repository SHACL validation*

8.8.2.10 SPARQL templates

Use the SPARQL template management API to create, edit, delete, and execute SPARQL templates, as well as to view the currently stored templates and their configurations.

See also:

curl examples for SPARQL template management

8.8.2.11 SQL views management

Use the SQL views management API to access, create, and edit SQL views tables. You can also use it to delete existing saved queries and to view the SQL views for the active repository.

See also:

curl examples for SQL views management

8.8.2.12 Saved queries

Use the saved queries API to create, edit and remove saved queries.

See also:

curl examples for saved queries

8.8.2.13 Security management

Use the security management API to integrate Workbench security into a system by enabling and disabling security and free access and by adding, editing, and removing user accounts.

See also:

curl examples for security management

8.8.3 Using GraphDB with the RDF4J API

This section describes how to use the RDF4J API to create and access GraphDB repositories, both on the local file system and remotely via the RDF4J HTTP server.

RDF4J is made up of a large collection of libraries, utilities and APIs. The important components for this section are:

- the RDF4J classes and interfaces (API), which provide a uniform access to the SAIL components from multiple vendors/publishers
- the RDF4J server application

8.8.3.1 RDF4J API

Programmatically, you can use GraphDB via the RDF4J Java framework of classes and interfaces. Documentation for these interfaces (including Javadoc) is available on the RDF4J project website. Code snippets in the sections below are taken from, or are variations of, the `developer-getting-started` examples that come in the `examples` directory of the *GraphDB distribution*.

Accessing a local repository

RDF4J 2 represents repository configurations as RDF graphs. A particular repository configuration is described as a resource, possibly a blank node, of type `http://www.openrdf.org/config/repository#Repository`.

This resource has an ID, a label, and an implementation, which in turn has a type, SAIL type, and relevant settings. A short repository configuration is taken from the `developer-getting-started` template file `repo-defaults.ttl`.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rep: <http://www.openrdf.org/config/repository#>.
@prefix sr: <http://www.openrdf.org/config/repository/sail#>.
@prefix sail: <http://www.openrdf.org/config/sail#>.
@prefix graphdb: <http://www.ontotext.com/config/graphdb#>.

[] a rep:Repository ;
  rep:repositoryID "graphdb-repo" ;
  rdfs:label "GraphDB Getting Started" ;
  rep:repositoryImpl [
    rep:repositoryType "graphdb:SailRepository" ;
    sr:sailImpl [
      sail:sailType "graphdb:Sail" ;
      graphdb:ruleset "owl-horst-optimized" ;
      graphdb:storage-folder "storage" ;
      graphdb:base-URL "http://example.org/" ;
      rep:repositoryType "file-repository" ;
      graphdb:imports "./ontology/owl.rdfs" ;
      graphdb:defaultNS "http://example.org/"
    ]
  ].
```

The following Java code uses the configuration to instantiate a repository and get a connection to it:

```
// Instantiate a local repository manager and initialize it
RepositoryManager repositoryManager = new LocalRepositoryManager(new File("."));
repositoryManager.initialize();

// Instantiate a repository graph model
TreeModel graph = new TreeModel();

// Read repository configuration file
InputStream config = getClass().class.getResourceAsStream("/repo-defaults.ttl");
RDFParser rdfParser = Rio.createParser(RDFFormat.TURTLE);
rdfParser.setRDFHandler(new StatementCollector(graph));
rdfParser.parse(config, RepositoryConfigSchema.NAMESPACE);
config.close();

// Retrieve the repository node as a resource
Resource repositoryNode = GraphUtil.getUniqueSubject(graph, RDF.TYPE, RepositoryConfigSchema.
↪REPOSITORY);

// Create a repository configuration object and add it to the repositoryManager
RepositoryConfig repositoryConfig = RepositoryConfig.create(graph, repositoryNode);
repositoryManager.addRepositoryConfig(repositoryConfig);

// Get the repository from repository manager, note the repository id set in configuration .ttl file
Repository repository = repositoryManager.getRepository("graphdb-repo");

// Open a connection to this repository
RepositoryConnection repositoryConnection = repository.getConnection();

// ... use the repository

// Shutdown connection, repository and manager
repositoryConnection.close();
repository.shutDown();
repositoryManager.shutDown();
```

The sample performs three basic steps:

1. Instantiate a local repository manager with the data directory to use for the repository storage files. (Repositories store their data in their own subdirectory from here.)
2. Add a repository configuration for the desired repository type to the manager.
3. 'Get' the repository and open a connection to it.

From then on, most activities will use the connection object to perform interactions such as executing queries, adding statements, committing transactions, and counting statements. See the `developer-getting-started` examples.

Note: The example above assumes that GraphDB Free edition is used. If you are using the Standard or Enterprise editions, set the system property `graphdb.license.file` to a valid license file.

Accessing a remote repository

The RDF4J server is a Web application that allows interaction with repositories using the HTTP protocol. It runs in a JEE compliant servlet container such as Tomcat and allows client applications to interact with repositories located on remote machines. In order to connect to and use a remote repository, you must replace the local repository manager with a remote one. The URL of the RDF4J server must be provided, but you don't need a repository configuration if the repository already exists on the server. The following lines can be added to the `developer-getting-started` example program, although a correct URL must be specified:

```
RepositoryManager repositoryManager =
        new RemoteRepositoryManager( "http://192.168.1.25:7200" );
repositoryManager.initialize();
```

The rest of the example program should work as expected, although the following library files must be added to the class-path:

- `commons-httpclient-3.1.jar`
- `commons-codec-1.10.jar`

8.8.4 GraphDB Plugin API

The GraphDB Plugin API is a framework and a set of public classes and interfaces that allow developers to extend GraphDB in many useful ways. These extensions are bundled into plugins, which GraphDB discovers during its initialization phase and then uses to delegate parts of its query or update processing tasks. The plugins are given low-level access to the GraphDB repository data, which enables them to do their job efficiently. They are discovered via the Java service discovery mechanism, which enables dynamic addition and removal of plugins from the system with no need to recompile GraphDB or change any configuration files.

8.8.4.1 A Java GraphDB plugin

A GraphDB plugin is a Java class that implements the `com.ontotext.trree.sdk.Plugin` interface. All public classes and interfaces of the plugin API are located in this `com.ontotext.trree.sdk` Java package. Here is what the plugin interface looks like in an abbreviated form:

```
/**
 * The base interface for a GraphDB plugin. As a minimum a plugin must implement this interface.
 * <p>
 * Plugins also need to be listed in META-INF/services/com.ontotext.trree.sdk.Plugin so that Java's
↪services
 * mechanism may discover them automatically.
 */
public interface Plugin extends Service {
    /**
     * A method used by the plugin framework to configure each plugin's file system directory. This
     * directory should be used by the plugin to store its files
     *
     * @param dataDir file system directory to be used for plugin related files
     */
    void setDataDir(File dataDir);

    /**
     * A method used by the plugin framework to provide plugins with a {@link Logger} object
     *
     * @param logger {@link Logger} object to be used for logging
     */
    void setLogger(Logger logger);

    /**
```

(continues on next page)

(continued from previous page)

```
     * Plugin initialization method called once when the repository is being initialized, after the plugin
↪has been
     * configured and before it is actually used. It enables plugins to execute whatever
     * initialization routines they consider appropriate, load resources, open connections, etc., based on
↪the
     * specific reason for initialization, e.g., backup.
     * <p>
     * The provided {@link PluginConnection} instance may be used to create entities needed by the plugin.
     *
     * @param reason           the reason for initialization
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void initialize(InitReason reason, PluginConnection pluginConnection);

    /**
     * Sets a new plugin fingerprint.
     * Every plugin should maintain a fingerprint of its data that could be used by GraphDB to determine
↪if the
     * data has changed or not. Initially, on system initialization the plugins are injected their
     * fingerprints as they reported them before the last system shutdown
     *
     * @param fingerprint the last known plugin fingerprint
     */
    void setFingerprint(long fingerprint);

    /**
     * Returns the fingerprint of the plugin.
     * <p>
     * Every plugin should maintain a fingerprint of its data that could be used by GraphDB to determine
↪if the
     * data has changed or not. The plugin fingerprint will become part of the repository fingerprint.
     *
     * @return the current plugin fingerprint based on its data
     */
    long getFingerprint();

    /**
     * Plugin shutdown method that is called when the repository is being shutdown. It enables plugins to
↪execute whatever
     * finalization routines they consider appropriate, free resources, buffered streams, etc., based on
↪the
     * specific reason for shutdown.
     *
     * @param reason the reason for shutdown
     */
    void shutdown(ShutdownReason reason);
}
```

Because it derives from the `Service` interface, the plugin is automatically discovered at run time, provided that the following conditions also hold:

- The plugin class is located in the classpath.
- It is mentioned in a `META-INF/services/com.ontotext.trree.sdk.Plugin` file in the classpath or in a `.jar` that is in the classpath. The full class signature must be written on a separate line in one of these files.

The only method introduced by the `Service` interface is `getName()`, which provides the name of the plugin (service). This name must be unique within a particular GraphDB repository and serves as a plugin identifier that you can use at any time to retrieve a reference to the plugin instance.

```
/**
 * Interface implemented by all run-time discoverable services (e.g. {@link Plugin} instances). Classes
```

(continues on next page)

(continued from previous page)

```
 * implementing this interface should furthermore be declared in the respective
 * META-INF/services/&lt;class.signature&gt; file and will then be discoverable at run-time.
 * <p>
 * Plugins need not implement this interface directly but rather implement {@link Plugin}.
 */
public interface Service {
    /**
     * Gets the service name (serves as a key for discovering the service)
     *
     * @return service name
     */
    String getName();
}
```

There are many more functions (interfaces) that a plugin could implement, but these are all optional and are declared in separate interfaces. Implementing any such complementary interface is the means to announce to the system what this particular plugin can do in addition to its mandatory plugin responsibilities. It is then automatically used as appropriate. See *List of plugin interfaces and classes*.

8.8.4.2 The life cycle of a plugin

A plugin's life cycle consists of five phases:

1. Discovery
2. Configuration
3. Initialization
4. Request Processing
5. Shutdown

The following describes these in more detail.

Discovery

This phase is executed at repository initialization. GraphDB searches for all plugin services in the classpath registered in the `META-INF/services/com.ontotext.trree.sdk.Plugins` service registry files, and then constructs a single instance of each plugin found.

Configuration

Every plugin instance discovered and constructed during the previous phase is then configured. During this phase, plugins are injected with a `Logger` object, which they use for logging (`setLogger(Logger logger)`), and the path to their own data directory (`setDataDir(File dataDir)`), which they create, if needed, and then use to store their data. If a plugin does not need to store anything to the disk, it can skip the creation of its data directory. However, if it needs to use this, it is guaranteed that this directory will be unique and available only to the particular plugin that it was assigned to.

This phase is also called when a plugin is enabled after repository initialization.

Initialization

After a plugin has been configured, the framework calls its `initialize(InitReason reason, PluginConnection pluginConnection)` method so that it gets the chance to do whatever initialization work is necessary. The passed instance of `PluginConnection` provides access to various other structures and interfaces, such as `Statements` and `Entities` instances (*Repository internals*), and a `SystemProperties` instance, which gives the plugins access to the system-wide configuration options and settings. Plugins typically use this phase to create IRIs that will be used to communicate with the plugin.

This phase is also called when a plugin is enabled after repository initialization.

Request processing

The plugin participates in the request processing. The request phase applies to the evaluation of SPARQL queries, `getStatements` calls, the transaction stages, and the execution of SPARQL updates. Various event notifications can also be part of this phase.

This phase is optional for the plugins but no plugin is useful without implementing at least one of *its interfaces*.

Request processing can be divided roughly into query processing and update processing.

Query processing

Query processing includes several sub-phases that can be used on their own or combined together:

- **Pre-processing**: Plugins are given the chance to modify the request before it is processed. In this phase, they could also initialize a context object, which will be visible until the end of the request processing (*Pre-processing*).
- **Pattern interpretation**: Plugins can choose to provide results for requested statement patterns (*Pattern interpretation*). This sub-phase applies only to queries.
- **Post-processing**: Before the request results are returned to the client, plugins are given a chance to modify them, filter them out, or even insert new results (*Post-processing*);

Update processing

Update processing includes several layers of processing:

- **Transaction events**: Plugins are notified about the beginning and end of a transaction.
- **Update handling**: Plugins can choose to handle certain updates (additions or removals) instead of letting the repository handle the updates as regular data.
- **Entities and statements notifications**: Plugins can be notified about the creation of entities and the addition and removal of statements.

Shutdown

During repository shutdown, each plugin is prompted to execute its own shutdown routines such as freeing up resources and flushing data to disk. This must be done in the `shutdown(ShutdownReason reason)` method.

This phase is also called when a plugin is disabled after repository initialization.

8.8.4.3 Repository internals

You can access the repository internals with an instance of `PluginConnection`:

```
/**
 * The {@link PluginConnection} interface provides access to various objects that can be used to query data
 * or get the properties of the current transaction. An instance of {@link PluginConnection} will be
↪passed to almost
 * all methods that a plugin may implement.
 */
public interface PluginConnection {
    /**
     * Returns an instance of {@link Entities} that can be used to retrieve or create RDF entities.
     *
     * @return an {@link Entities} instance
     */
    Entities getEntities();

    /**
     * Returns an instance of {@link Statements} that can be used to retrieve RDF statements.
     *
     * @return a {@link Statements} instance
     */
    Statements getStatements();

    /**
     * Returns an instance of {@link Repository} that can be used for higher level access to the
↪repository.
     *
     * @return a {@link Repository} instance
     */
    Repository getRepository();

    /**
     * Returns the transaction ID of the current transaction or 0 if no explicit transaction is available.
     *
     * @return the transaction ID
     */
    long getTransactionId();

    /**
     * Returns the update testing status. In a multi-node GraphDB configuration (currently only GraphDB
↪EE) an update
     * will be sent to multiple nodes. The first node that receives the update will be used to test if the
↪update is
     * successful and only if so, it will be send to other nodes. Plugins may use the update test status
↪to perform
     * certain operations only when the update is tested (e.g. indexing data via an external service). The
↪method will
     * return true if this is a GraphDB EE worker node testing the update or this is GraphDB Free or SE.
↪The method will
     * return false only if this is a GraphDB EE worker node that is receiving a copy of the original
↪update
     * (after successful testing on another node).
     *
     * @return true if this update is sent for the first time (testing the update), false otherwise
     */
    boolean isTesting();

    /**
     * Returns an instance of {@link SystemProperties} that can be used to retrieve various properties
↪that identify
```

(continues on next page)

(continued from previous page)

```
     * the current GraphDB installation and repository.
     *
     * @return an instance of {@link SystemProperties}
     */
    SystemProperties getProperties();

    /**
     * Returns the repository fingerprint. Note that during an active transactions the fingerprint will be
↪updated
     * at the very end of the transaction. Call it in {@link com.ontotext.trree.sdk.
↪PluginTransactionListener#transactionCompleted(PluginConnection)}
     * if you want to get the updated fingerprint for the just-completed transaction.
     *
     * @return the repository fingerprint
     */
    String getFingerprint();

    /**
     * Returns whether the current GraphDB instance is part of a cluster. This is useful in cases where a
↪plugin may modify
     * the fingerprint via a query. To protect cluster integrity the fingerprint may be changed only via
↪an update.
     *
     * @return true if the current instance is in cluster group, false otherwise
     */
    boolean isInCluster();

    /*
     * Creates a thread-safe instance of this {@link PluginConnection} that can be used by other threads.
     * Note that every {@link ThreadsafePluginConnecton} must be explicitly closed when no longer needed.
     *
     * @return an instance of {@link ThreadsafePluginConnecton}
     */
    ThreadsafePluginConnecton getThreadsafeConnection();

    /**
     * Returns an instance of {@link SecurityContext} that can be used to check if the user that initiated
↪a plugin
     * request has the required access level.
     *
     * @return an instance of {@link SecurityContext}
     */
    SecurityContext getSecurityContext();
}
```

`PluginConnection` instances passed to the plugin are not thread-safe and not guaranteed to operate normally once the called method returns. If the plugin needs to process data asynchronously in another thread it must get an instance of `ThreadsafePluginConnection` via `PluginConnection.getThreadsafeConnection()`. Once the allocated thread-safe connection is no longer needed it **should be closed**.

`PluginConnection` provides access to various other interfaces that access the repository's data (`Statements` and `Entities`), the current transaction's properties, the repository fingerprint, various system and repository properties (`SystemProperties`), and the security context of plugin requests (`SecurityContext`).

`PluginConnection` also provides higher level access to the repository via the `Repository` interface, with the ability for simple data updates.

Statements and entities

In order to enable efficient request processing, plugins are given low-level access to the repository data and internals. This is done through the `Statements` and `Entities` interfaces.

The `Entities` interface represents a set of RDF objects: IRIs, blank nodes, literals, and RDF-star embedded triples. All such objects are termed *entities* and are given unique `long` identifiers. The `Entities` instance is responsible for resolving these objects from their identifiers and inversely for looking up the identifier of a given entity. Most plugins process entities using their identifiers, because dealing with integer identifiers is a lot more efficient than working with the actual RDF entities they represent. The `Entities` interface is the single entry point available to plugins for entity management. It supports the addition of new entities, look-up of entity types and properties, resolving entities, and more.

It is possible to declare two RDF objects to be equivalent in a GraphDB repository — for example, by using *owl:sameAs optimization*. In order to provide a way to use such declarations, the `Entities` interface assigns a class identifier to each entity. For newly created entities, this class identifier is the same as the entity identifier. When two entities are declared equivalent, one of them adopts the class identifier of the other, and thus they become members of the same equivalence class. The `Entities` interface exposes the entity class identifier for plugins to determine which entities are equivalent.

Entities within an `Entities` instance have a certain scope. There are three entity scopes:

- **Default** — entities are persisted on the disk and can be used in statements that are also physically stored on disk. They have positive (non-zero) identifiers and are often referred to as physical or data entities.
- **System** — system entities have negative identifiers and are not persisted on the disk. They can be used, for example, for system (or *magic*) predicates that can provide configuration to a plugin or request something to be handled by a plugin. They are available throughout the whole repository lifetime, but after restart, they must be recreated again.
- **Request** — entities are not persisted on disk and have negative identifiers. They only live in the scope of a particular request, and are not visible to other concurrent requests. These entities disappear immediately after the request processing finishes. The request scope is useful for temporary entities such as those entities that are returned by a plugin as a response to a particular query.

The `Statements` interface represents a set of RDF statements, where a statement is a quadruple of subject, predicate, object, and context RDF entity identifiers. You can search for statements but not modify them.

Consuming or returning statements

An important abstract class, which is related to GraphDB internals, is `StatementIterator`. It has a `boolean next()` method, which attempts to scroll the iterator onto the next available statement and returns `true` only if it succeeds. In case of success, its `subject`, `predicate`, `object`, and `context` fields are initialized with the respective components of the next statement. Some properties of each statement are also available via the following methods:

- `boolean isReadOnly()` — returns `true` if the statement is in the Axioms part of the rule-file or is imported at initialization
- `boolean isExplicit()` — returns `true` if the statement is explicitly asserted
- `boolean isImplicit()` — returns `true` if the statement is produced by the inferencer (raw statements can be both explicit and implicit)

The following brief example puts `Statements`, `Entities`, and `StatementIterator` together in order to output all literals that are related to a given URI:

```
// resolve the URI identifier
long id = entities.resolve(SimpleValueFactory.getInstance().createIRI("http://example/uri"));

// retrieve all statements with this identifier in subject position
StatementIterator iter = statements.get(id, 0, 0, 0);
while (iter.next()) {
```

(continues on next page)

(continued from previous page)

```
    // only process literal objects
    if (entities.getType(iter.object) == Entities.Type.LITERAL) {
        // resolve the literal and print out its value
        Value literal = entities.get(iter.object);
        System.out.println(literal.stringValue());
    }
}
```

`StatementIterator` is also used to return statements via one of *the pattern interpretation interfaces*.

Each GraphDB transaction has several properties accessible via `PluginConnection`:

Transaction ID (`PluginConnection.getTransactionId()`) An integer value. Bigger values indicate newer transactions.

Testing (`PluginConnection.isTesting()`) A boolean value indicating the testing status of a transaction. In GraphDB EE the testing transaction is the first execution of a given transaction that determines if the transaction can be executed successfully before being propagated to the entire cluster. Despite the `_testing_` name it is a full-featured transaction that will modify the data. In GraphDB Free and SE the transaction is always executed only once so it is always testing there.

Repository access

`PluginConnection` provides higher level access to the repository via `getRepository()`.

This higher level access to the repository implements simple add and remove statement operations. You can use it to modify the data stored in the repository while a transaction is active.

The `getRepository()` method returns an instance of `Repository` (note that this is not an RDF4J repository instance):

```
/**
 * Interface that provides higher level access to the repository, including the ability to add and remove
↪statements.
 *
 * @since 10
 */
public interface Repository {
    /**
     * Returns true if this instance is allowed to add statements to the repository.
     * Adding statements is disallowed during plugin initialization, without an active transaction, and in
↪thread-safe
     * instances obtained via {@link PluginConnection#getThreadsafeConnection()}.
     *
     * @return true if adding is allowed, false otherwise.
     */
    boolean isAddAllowed();

    /**
     * Returns true if this instance is allowed to remove statements from the repository.
     * Removing statements is disallowed during plugin initialization, without an active transaction,
↪during a parallel
     * load, and in thread-safe instances obtained via {@link PluginConnection#getThreadsafeConnection()}.
     *
     * @return true if adding is allowed, false otherwise.
     */
    boolean isRemoveAllowed();

    /**
     * Add a statement to the repository.
     *
```

(continues on next page)

(continued from previous page)

```
     * @param subject   subject of the statement to add
     * @param predicate predicate of the statement to add
     * @param object    object of the statement to add
     * @param contexts  context(s) to add the statement to, if no contexts are specified, the statement
     *                  will be added to the default graph.
     * @throws IllegalStateException if this instance isn't allowed to add statements
     */
    void addStatement(Resource subject, IRI predicate, Value object, Resource... contexts)
            throws IllegalStateException;

    /**
     * Removes all statements matching the specified subject, predicate and object from the repository.
     * All three parameters may be null to indicate wildcards.
     *
     * @param subject   subject of the statement to remove
     * @param predicate predicate of the statement to remove
     * @param object    object of the statement to remove
     * @param contexts  context(s) to remove the statement from, if no contexts are specified, the
↪statement
     *                  will be removed from all graphs. Use null to remove from the default graph only.
     * @throws IllegalStateException if this instance isn't allowed to remove statements
     */
    void removeStatements(Resource subject, IRI predicate, Value object, Resource... contexts)
            throws IllegalStateException;
}
```

System properties

`PluginConnection` provides access to various static repository and system properties via `getProperties()`. Most of the values of these properties are set at repository initialization time and will not change while the repository is operating. The values for the product type and capabilities may change after repository initialization if the GraphDB license is updated.

The `getProperties()` method returns an instance of `SystemProperties`:

```
/**
 * This interface represents various properties for the running GraphDB instance and the repository as
↪seen by the Plugin API.
 */
public interface SystemProperties {
    /**
     * Returns the read-only status of the current repository.
     *
     * @return true if read-only, false otherwise
     */
    boolean isReadOnly();

    /**
     * Returns the number of bits needed to represent an entity id
     *
     * @return the number of bits as an integer
     */
    int getEntityIdSize();

    /**
     * Returns the product type of the current GraphDB license.
     *
     * @return one of {@link ProductType#FREE}, {@link ProductType#SE} or {@link ProductType#EE}
     */
```

(continues on next page)

(continued from previous page)

```
    ProductType getProductType();

    /**
     * Checks whether the current license has the provided product capability.
     *
     * @param productCapability a product capability
     * @return true if the capability is supported by the license,  false otherwise.
     */
    boolean hasProductCapability(String productCapability);

    /**
     * Returns the full GraphDB version string.
     *
     * @return a string describing the GraphDB version
     */
    String getVersion();

    /**
     * Returns the GraphDB major version component.
     *
     * @return the major version as an integer
     */
    int getVersionMajor();

    /**
     * Returns the GraphDB minor version component.
     *
     * @return the minor version as an integer
     */
    int getVersionMinor();

    /**
     * Returns the GraphDB patch version component.
     *
     * @return the patch version as an integer
     */
    int getVersionPatch();

    /**
     * Returns the number of cores in the currently set license up to the physical number of cores on the
↪machine.
     *
     * @return the number of cores as an integer
     */
    int getNumberOfLicensedCores();

    /**
     * Retrieve string repository configuration identified by the given IRI.
     *
     * @param settingName  the configuration identifier
     * @param defaultValue the default value to return if not configured
     * @return the configuration value or default value
     */
    String getRepositorySetting(IRI settingName, String defaultValue);

    /**
     * Retrieve boolean repository configuration identified by the given IRI.
     *
     * @param settingName  the configuration identifier
     * @param defaultValue the default value to return if not configured
     * @return the configuration value or default value
```

(continues on next page)

(continued from previous page)

```
     */
    boolean getRepositorySetting(IRI settingName, boolean defaultValue);

    /**
     * Retrieve integer repository configuration identified by the given IRI.
     *
     * @param settingName  the configuration identifier
     * @param defaultValue the default value to return if not configured
     * @return the configuration value or default value
     */
    int getRepositorySetting(IRI settingName, int defaultValue);

    /**
     * Retrieve multi-valued string based repository configuration identified by the given IRI.
     *
     * @param settingName  the configuration identifier
     * @return the configuration value or empty array
     */
    String[] getRepositorySetting(IRI settingName);

    /**
     * The possible product types of the installed GraphDB license.
     */
    enum ProductType {
        /**
         * GraphDB Free repository
         */
        FREE,
        /**
         * GraphDB SE repository
         */
        SE,
        /**
         * GraphDB EE worker repository
         */
        EE
    }
}
```

Repository properties

There are some dynamic repository properties that may change once a repository has been initialized:

Repository fingerprint (`PluginConnection.getFingerprint()`) The repository fingerprint. Note that the fingerprint will be updated at the very end of a transaction so the updated fingerprint after a transaction should be accessed within `PluginTransactionListener.transactionCompleted()`.

Whether the repository is attached to a cluster (`PluginConnection.isAttached()`) GraphDB EE worker repositories are typically attached to a master repository and not accessed directly. When this is the case this method will return `true` and the plugin may use it to refuse to perform actions that may cause the fingerprint to change outside of a transaction. In GraphDB Free and SE the method always returns `false`.

Security context

`PluginConnection` provides access to the security context of plugin requests via `getSecurityContext()`.

The security context can be used to check if the user that initiated a plugin request has the required access level based on simple criteria such as having write access to the repository, checking if the user has a specific role or username that matches an access control list maintained by the plugin.

The `getSecurityContext()` method returns an instance of `SecurityContext`:

```
/**
 * Plugin interface that provides access to the security context.
 */
public interface SecurityContext {
    /**
     * Returns the username of the user that initiated the plugin request.
     *
     * @return a username
     */
    String getUsername();

    /**
     * Returns true if the user that initiated the plugin request has write access to the repository.
     *
     * @return true if write granted, false otherwise
     */
    boolean hasWriteAccess();

    /**
     * Returns the roles of the user that initiated the plugin request.
     *
     * @return a set of user roles
     */
    Set<String> getRoles();

    /**
     * Returns true if the user that initiated the plugin request has the supplied role.
     *
     * @return true if the user has the role, false otherwise
     */
    boolean hasRole(String role);
}
```

8.8.4.4 Query processing

As mentioned above, a plugin's interaction with each of the request-processing phases is optional. The plugin declares if it plans to participate in any phase by implementing the appropriate interface.

Pre-processing

A plugin that will pre-process requests must implement the `Preprocessor` interface. It looks like this:

```
/**
 * Interface that should be implemented by all plugins that need to maintain per-query context.
 */
public interface Preprocessor {
    /**
     * Pre-processing method called once for every SPARQL query or getStatements() request before it is
     * processed.
```

(continues on next page)

(continued from previous page)

```
     *
     * @param request request object
     * @return context object that will be passed to all other plugin methods in the future stages of the
     * request processing
     */
    RequestContext preprocess(Request request);
}
```

The `preprocess(Request request)` method receives the request object and returns a `RequestContext` instance. The passed request parameter is an instance of one the interfaces extending `Request`, depending on the type of the request (`QueryRequest` for a SPARQL query or `StatementRequest` for "get statements"). The plugin changes the request object accordingly and initializes and and returns its context object, which is passed back to it in every other method during the request processing phase. The returned request context may be `null`, but regardless of this, it is only visible to the plugin that initializes it. It can be used to store data visible for (and only for) this whole request, for example, to pass data related to two different statement patterns recognized by the plugin. The request context gives further request processing phases access to the `Request` object reference. Plugins that opt to skip this phase do not have a request context and cannot get access to the original `Request` object.

Plugins may create their own `RequestContext` implementation or use the default one, `RequestContextImpl`.

Pattern interpretation

This is one of the most important phases in the life cycle of a plugin. Most plugins need to participate in it. This is the point where request statement patterns must get evaluated and statement results are returned.

For example, consider the following SPARQL query:

```
SELECT * WHERE {
    ?s <http://example.com/predicate> ?o
}
```

There is just one statement pattern inside this query: `?s <http://example/predicate> ?o`. All plugins that have implemented the `PatternInterpreter` interface (thus declaring that they intend to participate in the pattern interpretation phase) are asked if they can interpret this pattern. The first one to accept it and return results will be used. If no plugin interprets the pattern, it will look to use the repository's physical statements — that is, the ones persisted on the disk.

Here is the `PatternInterpreter` interface:

```
/**
 * Interface implemented by plugins that want to interpret basic triple patterns
 */
public interface PatternInterpreter {
    /**
     * Estimate the number of results that could be returned by the plugin for the given parameters
     *
     * @param subject          subject ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param object           object ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or null
     * @return approximate number of results that could potentially be returned for this parameters by the
     * interpret() method
     */
    double estimate(long subject, long predicate, long object, long context, PluginConnection
↪pluginConnection,
```

(continues on next page)

(continued from previous page)

```
                    RequestContext requestContext);

    /**
     * Interpret basic triple pattern and return {@link StatementIterator} with results
     *
     * @param subject        subject ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param predicate      predicate ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param object         object ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param context        context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext context object as returned by {@code Preprocessor.preprocess()} or null
     * @return statement iterator of results
     */
    StatementIterator interpret(long subject, long predicate, long object, long context,
                                PluginConnection pluginConnection, RequestContext requestContext);
}
```

The `estimate()` and `interpret()` methods take the same arguments and are used in the following way:

- Given a statement pattern like the one in the SPARQL query above, all plugins that implement `PatternInterpreter` are asked to `interpret()` the pattern. The `subject`, `predicate`, `object` and `context` values are either the identifiers of the values in the pattern or 0 if any of them is an unbound variable. The `statements` and `entities` objects represent respectively the statements and entities that are available for this particular request. For instance, if the query contains any `FROM <http://some/graph>` clauses, the `statements` object will only provide access to the statements in the defined named graphs. Similarly, the `entities` object contains entities that might be valid only for this particular request. The plugin's `interpret()` method must return a `StatementIterator` if it intends to interpret this pattern or `null` if it refuses.
- In case the plugin signals that it will interpret the given pattern (returns a non-`null` value), GraphDB's query optimizer will call the plugin's `estimate()` method in order to get an estimate of how many results will be returned by the `StatementIterator` returned by `interpret()`. This estimate does not need to be precise. But the more precise it is, the more likely the optimizer will make an efficient optimization.

  There is a slight difference in the values that will be passed to `estimate()`. The statement components such as `subject` might not only be entity identifiers, but they can also be set to 2 special values:

  - `Entities.BOUND` — the pattern component is said to be bound, but its particular binding is not yet known.
  - `Entities.UNBOUND` — the pattern component will not be bound. These values must be treated as hints to the `estimate()` method to provide a better approximation of the result set size, although its precise value cannot be determined before the query is actually run.
- After the query has been optimized, the `interpret()` method of the plugin might be called again should any variable become bound due to the pattern reordering applied by the optimizer. Plugins must be prepared to expect different combinations of bound and unbound statement pattern components and return appropriate iterators.

The `requestContext` parameter is the value returned by the `preprocess()` method if one exists, or `null` otherwise.

Results are *returned as statements*.

The plugin framework also supports the interpretation of an extended type of a *list* pattern. Consider the following SPARQL queries:

```
SELECT * WHERE {
    ?s <http://example.com/predicate> (?o1 ?o2)
}
```

```
SELECT * WHERE {
    (?s1, ?s2) <http://example.com/predicate> ?o
}
```

Internally the object or subject list will be converted to a series of triples conforming to rdf:List. These triples can be handled with `PatternInterpreter` but the whole list semantics will have to be implemented by the plugin.

In order to make this task easier the Plugin API defines two additional interfaces very similar to the `PatternInterpreter` interface — `ListPatternInterpreter` and `SubjectListPatternInterpreter`.

`ListPatternInterpreter` handles lists in the *object* position:

```
/**
 * Interface implemented by plugins that want to interpret list-like triple patterns
 */
public interface ListPatternInterpreter {
    /**
     * Estimate the number of results that could be returned by the plugin for the given parameters
     *
     * @param subject          subject ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param objects          object IDs (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or null
     * @return approximate number of results that could potentially be returned for this parameters by the
     * interpret() method
     */
    double estimate(long subject, long predicate, long[] objects, long context, PluginConnection
↪pluginConnection,
                    RequestContext requestContext);

    /**
     * Interpret list-like triple pattern and return {@link StatementIterator} with results
     *
     * @param subject          subject ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param objects          object IDs (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or null
     * @return statement iterator of results
     */
    StatementIterator interpret(long subject, long predicate, long[] objects, long context,
                                PluginConnection pluginConnection, RequestContext requestContext);
}
```

It differs from `PatternInterpreter` by having multiple objects passed as an array of `long`, instead of a single `long` object. The semantics of both methods are equivalent to the one in the basic pattern interpretation case.

`SubjectListPatternInterpreter` handles lists in the *subject* position:

```
/**
 * Interface implemented by plugins that want to interpret list-like triple patterns
 */
```

(continues on next page)

(continued from previous page)

```
public interface SubjectListPatternInterpreter {
    /**
     * Estimate the number of results that could be returned by the plugin for the given parameters
     *
     * @param subjects         subject IDs (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param object           object ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or null
     * @return approximate number of results that could potentially be returned for this parameters by the
     * interpret() method
     */
    double estimate(long[] subjects, long predicate, long object, long context, PluginConnection
↪pluginConnection,
                    RequestContext requestContext);

    /**
     * Interpret list-like triple pattern and return {@link StatementIterator} with results
     *
     * @param subjects       subject IDs (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param predicate      predicate ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND}
↪)
     * @param object         object ID (alternatively {@link Entities#BOUND} or {@link Entities#UNBOUND})
     * @param context        context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext context object as returned by {@code Preprocessor.preprocess()} or null
     * @return statement iterator of results
     */
    StatementIterator interpret(long[] subjects, long predicate, long object, long context,
                                PluginConnection pluginConnection, RequestContext requestContext);
}
```

It differs from `PatternInterpreter` by having multiple subjects passed as an array of `long`, instead of a single `long` subject. The semantics of both methods are equivalent to the one in the basic pattern interpretation case.

Post-processing

There are cases when a plugin would like to modify or otherwise filter the final results of a request. This is where the `Postprocessor` interface comes into play:

```
/**
 * Interface that should be implemented by plugins that need to post-process results from queries.
 */
public interface Postprocessor {
    /**
     * A query method that is used by the framework to determine if a {@link Postprocessor} plugin really
↪wants to
     * post-process the request results.
     *
     * @param requestContext the request context reference
     * @return boolean value
     */
    boolean shouldPostprocess(RequestContext requestContext);
```

(continues on next page)

(continued from previous page)

```
    /**
     * Method called for each {@link BindingSet} in the query result set. Each binding set is processed in
     * sequence by all plugins that implement the {@link Postprocessor} interface, piping the result
↪returned
     * by each plugin into the next one. If any of the post-processing plugins returns null the result is
     * deleted from the result set.
     *
     * @param bindingSet     binding set object to be post-processed
     * @param requestContext context objected as returned by {@link Preprocessor#preprocess(Request)} (in
↪case this plugin
     *                       implemented this interface)
     * @return binding set object that should be post-processed further by next post-processing plugins or
     * null if the current binding set should be deleted from the result set
     */
    BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext);

    /**
     * Method called after all post-processing has been finished for each plugin. This is the point where
     * every plugin could introduce its results even if the original result set was empty
     *
     * @param requestContext context objected as returned by {@link Preprocessor#preprocess(Request)} (in
↪case this plugin
     *                       implemented this interface)
     * @return iterator for resulting binding sets that need to be added to the final result set
     */
    Iterator<BindingSet> flush(RequestContext requestContext);
}
```

The `postprocess()` method is called for each binding set that is to be returned to the repository client. This method may modify the binding set and return it, or alternatively, return `null`, in which case the binding set is removed from the result set. After a binding set is processed by a plugin, the possibly modified binding set is passed to the next plugin that has post-processing enabled. After the binding set is processed by all plugins (in the case where no plugin deletes it), it is returned to the client. Finally, after all results are processed and returned, each plugin's `flush()` method is called to introduce new binding set results in the result set. These in turn are finally returned to the client.

8.8.4.5 Update processing

Updates involving specific predicates

Along with query and read processing, plugins can process update operations for statement patterns containing specific predicates. In order to intercept updates, a plugin must implement the `UpdateInterpreter` interface. During initialization, the framework calls `getPredicatesToListenFor()` once so that the plugin can indicate which predicates it is interested in.

From then onwards, the plugin framework filters updates for statements using these predicates and notifies the plugin. The plugin may do whatever processing is required and must return a boolean value indicating whether the statement should be skipped. Skipped statements are not processed further by GraphDB, so the insert or delete will have no effect on actual data in the repository.

```
/**
 * An interface that should be implemented by the plugins that want to be notified for particular update
 * events. The getPredicatesToListenFor() method should return the predicates of interest to the plugin.
↪This
 * method will be called once only immediately after the plugin has been initialized. After that point the
 * plugin's interpretUpdate() method will be called for each inserted or deleted statement sharing one of
↪the
 * predicates of interest to the plugin (those returned by getPredicatesToListenFor()).
 */
```

(continues on next page)

(continued from previous page)

```
public interface UpdateInterpreter {
    /**
     * Returns the predicates for which the plugin needs to get notified when a statement with such a
↪predicate is added or removed.
     *
     * @return array of predicates as entity IDs
     */
    long[] getPredicatesToListenFor();

    /**
     * Hook that is called whenever a statement containing one of the registered predicates
     * (see {@link #getPredicatesToListenFor()} is added or removed.
     *
     * @param subject          subject value of the updated statement
     * @param predicate        predicate value of the updated statement
     * @param object           object value of the updated statement
     * @param context          context value of the updated statement
     * @param isAddition       true if the statement was added, false if it was removed
     * @param isExplicit       true if the updated statement was explicit one
     * @param pluginConnection an instance of {@link PluginConnection}
     * @return true - when the statement was handled by the plugin only and should <i>NOT</i> be added to/
↪removed from the repository,
     * false - when the statement should be added to/removed from the repository
     */
    boolean interpretUpdate(long subject, long predicate, long object, long context, boolean isAddition,
                            boolean isExplicit, PluginConnection pluginConnection);
}
```

Removal of entire contexts

Statement deletion in GraphDB is specified as a quadruple *(subject, predicate, object, context)*, where each position can be explicit or null. Null in this case means all subjects, predicates, objects or contexts depending on the position where null was specified.

When at least one of the positions is non-null, the plugin framework will fire individual events for each matching and removed statement.

When all positions are null (that is, everything in the repository has been deleted) the operation will be optimized internally and individual events will not be fired. This means that `UpdateInterpreter` and `StatementListener` will not be called.

`ClearInterpreter` is an interface that allows plugins to detect the removal of entire contexts or removal of all data in the repository:

```
/**
 * This interface can be implemented by plugins that want to be notified on clear()
 * or remove() (all statements in any context).
 */
public interface ClearInterpreter {
    /**
     * Notification called before the statements are removed from the given context.
     *
     * @param context          the ID of the context or 0 if all contexts
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void beforeClear(long context, PluginConnection pluginConnection);

    /**
     * Notification called after the statements have been removed from the given context.
     *
```

(continues on next page)

(continued from previous page)

```
     * @param context          the ID of the context or 0 if all contexts
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void afterClear(long context, PluginConnection pluginConnection);
}
```

Intercepting data for specific contexts

The Plugin API provides a way to intercept data inserted into or removed from a particular predefined context. The `ContextUpdateHandler` interface:

```
/**
 * This interface provides a mechanism for plugins to handle updates to certain contexts.
 * When a plugin requests handling of a context, all data for that context will forwarded to the plugin
 * and not inserted into any GraphDB collections.
 * <p>
 * Note that unlike other plugin interfaces, {@link ContextUpdateHandler} does not use entity IDs but
↪works directly
 * with the RDF values. Data handled by this interface does not reach the entity pool and so no entity IDs
↪are created.
 */
public interface ContextUpdateHandler {
    /**
     * Returns the contexts for which the plugin will handle the updates.
     *
     * @return array of {@link Resource}
     */
    Resource[] getUpdateContexts();

    /**
     * Hook that handles updates for the configured contexts.
     *
     * @param subject          subject value of the updated statement
     * @param predicate        predicate value of the updated statement
     * @param object           object value of the updated statement
     * @param context          context value of the updated statement (can be null when not an addition,
↪then it means remove from all contexts)
     * @param isAddition       true if statement is being added, false if statement is being removed
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void handleContextUpdate(Resource subject, IRI predicate, Value object, Resource context, boolean
↪isAddition,
                             PluginConnection pluginConnection);
}
```

This is similar to *Updates involving specific predicates* with some important differences:

- `ContextUpdateHandler`
  - Configured via a list of **contexts** specified as **IRI objects**.
  - Statements with these contexts are passed to the plugin as **Value objects** and never enter any of the database collections.
  - The plugin is assumed to **always** handle the update.
- `UpdateInterpreter`
  - Configured via a list of **predicates** specified as **integer IDs**.
  - Statements with these predicates are passed to the plugin as **integer IDs** after their RDF values are converted to integer IDs in the entity pool.

- The plugin **decides** whether to handle the statement or pass it on to other plugins and eventually to the database.

This mechanism is especially useful for the creation of virtual contexts (graphs) whose data is stored within a plugin and never pollutes any of the database collections with unnecessary values.

Unlike the rest of the Plugin API this interface uses RDF values as objects bypassing the use of integer IDs.

8.8.4.6 Transactions

A plugin may require participation in the transaction workflow — for example, if it needs to update certain data structures such that they reflect the actual data in the repository. Without being part of the transaction the plugin would not know when to persist or discard a given state.

Transactions can be easily tracked by implementing the `PluginTransactionListener` interface:

```
/**
 * The {@link PluginTransactionListener} allows plugins to be notified about transactions (start,
↪commit+completed or abort)
 */
public interface PluginTransactionListener {
    /**
     * Notifies the listener about the start of a transaction.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionStarted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the commit phase of a transaction. Plugins should use this event to
↪perform their own
     * commit work if needed or to abort the transaction if needed.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionCommit(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the completion of a transaction. This will be the last event in a
↪successful transaction.
     * The plugin is not allowed to throw any exceptions here and if so they will be ignored. If a plugin
↪needs to abort
     * a transaction it should be done in {@link #transactionCommit(PluginConnection)}.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionCompleted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the abortion of a transaction. This will be the last event in an
↪aborted transaction.
     * <p>
     * Plugins should revert any modifications caused by this transaction, including the fingerprint.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionAborted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about a user abort request. A user abort request is a request by an end-user
↪to abort the
     * transaction. Unlike the other events this will be called asynchronously whenever the request is
↪received.
```

(continues on next page)

(continued from previous page)

```
     * <p>
     * Plugins may react and terminate any long-running computation or ignore the request. This is just a
↪handy way
     * to speed up abortion when a user requests it. For example, this event may be received
↪asynchronously while
     * the plugin is indexing data (in {@link #transactionCommit(PluginConnection)} running in the main
↪thread).
     * The plugin may notify itself that the indexing should stop. Regardless of the actions taken by the
↪plugin
     * the transaction may still be aborted and {@link #transactionAborted(PluginConnection)} will be
↪called.
     * All clean up of the abortion should be handled in {@link #transactionAborted(PluginConnection)}.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    default void transactionAbortedByUser(PluginConnection pluginConnection) {

    }
}
```

Each transaction has a beginning signalled by a call to `transactionStarted()`. Then the transaction can proceed in several ways:

- Commit and completion by calling:
  - `transactionCommit()`
  - `transactionCompleted()`
- Commit followed by abortion (typically because another plugin aborted the transaction in its own `transactionCommit()`) by calling:
  - `transactionCommit()`
  - `transactionAborted()`
- Abortion before entering commit by calling:
  - `transactionAborted()`

Plugins should strive to do all heavy transaction work in `transactionCommit()` in such a way that a call to `transactionAborted()` can revert the changes. Plugins may throw exceptions in `transactionCommit()` in order to abort the transaction — for example if some constraint was violated.

Plugins should do no heavy processing in `transactionCompleted()` and may not throw exceptions there. Such exceptions will be logged and ignored and the transaction will still go through normally.

The `transactionAbortedByUser()` will be called asynchronously (for example, while the plugin is executing `transactionCommit()` in the main update thread) when a user requests that the transaction be aborted. The plugin may use this to signal its other thread to abort processing at the earliest convenience or to simply ignore the request.

8.8.4.7 Exceptions

Plugins may throw exceptions on invalid input, constraint violations or unexpected events such as running out of disk space. It is possible to throw such exceptions almost everywhere with the notable exception of `PluginTransactionListener.transactionCompleted()`.

A good practice is to construct an instance of `PluginException` or one of its subclasses:

- `ClientErrorException` — for example when the user provided invalid input.
- `ServerErrorException` — for example when an unexpected server error occurred, such as lack of disk permissions.

8.8.4.8 Accessing other plugins

Plugins can use other plugins. For example, the Lucene-based full-text search plugin can make use of the rank values provided by the RDF Rank plugin to perform query result scoring and ordering. This is not a matter of re-using program code (for example, in a `.jar` with the common classes); it is about re-using data. The mechanism to do this allows plugins to obtain references to other plugin objects by knowing their names. To achieve this, they only need to implement the `PluginDependency` interface:

```
/**
 * Interface that should be implemented by plugins that depend on other plugins and want to be able to
 * retrieve references to them at runtime.
 */
public interface PluginDependency {
    /**
     * Method used by the plugin framework to inject a {@link PluginLocator} instance in the plugin.
     *
     * @param locator a {@link PluginLocator} instance
     */
    void setLocator(PluginLocator locator);
}
```

They are then injected into an instance of the `PluginLocator` interface (during the configuration phase), which does the actual plugin discovery for them:

```
/**
 * Interface that supports obtaining of a plugin instance by plugin name. An object implementing this
 * interface is injected into plugins that implement the {@link PluginDependency} interface.
 */
public interface PluginLocator {
    /**
     * Retrieves a {@link Plugin} instance by plugin name
     *
     * @param name name of the plugin
     * @return a {@link Plugin} instance or null if a plugin with that name is not available
     */
    Plugin locate(String name);

    /**
     * Retrieves a {@link RDFRankProvider} instance.
     *
     * @return a {@link RDFRankProvider} instance or null if no {@link RDFRankProvider} is available
     */
    RDFRankProvider locateRDFRankProvider();
}
```

Having a reference to another plugin is all that is needed to call its methods directly and make use of its services.

An important interface related to accessing other plugins is the `RDFRankProvider` interface. The sole implementation is the RDF Rank plugin but you can easily replace it with another implementation. By having a dedicated interface it is easy for plugins to get access to RDF rank values without relying on a specific implementation.

8.8.4.9 List of plugin interfaces and classes

Basics

`Plugin` The basic interface that defines a plugin.

`PluginBase` A reference abstract implementation of `Plugin` that can serve as the base for implementing plugins.

There are a couple of extensions of the `Plugin` interface that add additional configuration or behavior to plugins:

`ParallelPlugin` Marks a plugin as aware of parallel processing. The plugin will be injected into an instance of `PluginExecutorService` via `setExecutorService(PluginExecutorService executorService)`.

`PluginExecutorService` is a simplified version of Java's `ExecutorService` and provides an easy mechanism for plugins to schedule parallel tasks safely.

No open-source plugins use `ParallelPlugin`.

`StatelessPlugin` Marks a plugin as *stateless*. Stateless plugins do not contribute to the repository fingerprint and their fingerprint will not be queried.

This is suitable for plugins that are unimportant for query results or update executions such as plugins that are not typically used in the normal data flow.

Open-source plugins using `StatelessPlugin`:

- Autocomplete
- Notifications logger

On `initialize()` and `shutdown()` plugins receive an enum value, `InitReason` and `ShutdownReason` respectively, describing the reason why the plugin is being initialized or shut down.

`InitReason`

- `DEFAULT`: initialized as part of the repository initialization or the plugin was enabled
- `CREATED_BACKUP`: initialized after a shutdown for backup
- `RESTORED_FROM_BACKUP`: initialized after a shutdown for restore

`ShutdownReason`

- `DEFAULT`: shutdown as part of the repository shutdown or the plugin was disabled
- `CREATE_BACKUP`: shutdown before backup
- `RESTORE_FROM_BACKUP`: shutdown before restore

Plugins may use the reason to handle their own backup scenarios. In most cases it is unnecessary since the plugin's files will be backed up or restored together with the rest of the repository data.

Data structures

For more information, see *Repository internals*.

`PluginConnection` The main entry to repository internals. Passed to almost all methods in Plugin API interfaces.

`ThreadsafePluginConnection` Thread-safe version of `PluginConnection`. Requested explicitly from `PluginConnection` and must be explicitly closed when no longer needed.

Open-source plugins using `ThreadsafePluginConnection`:

- Autocomplete

`Entities` Provides access to the repository's entities. Entities are mappings from integer IDs to RDF values (IRIs, blank nodes, literals, and RDF-star embedded triples).

`Statements` Provides access to the repository's statements. Results are returned as `StatementIterator` instances.

`StatementIterator` Interface for returning statements. Used both by `Statements` to list repository data and by plugins to return data via *Pattern interpretation*.

`SystemProperties` Provides access to static repository and system properties such as the GraphDB version and repository type.

All open-source plugins use the repository internals.

Query request handlers

For more information, see *Query processing*.

Pattern interpretation handlers

The pattern interpretation handlers interpret the evaluation of triple patterns. Each triple pattern will be sent to plugins that implement the respective interface.

For more information, see *Pattern interpretation*.

`PatternInterpreter` Interprets a simple triple pattern where the subject, predicate, object and context are single values.

This interface handles *all* triple patterns: `subject predicate object context`.

Open-source plugins using `PatternInterpreter`:

- Autocomplete
- GeoSPARQL
- Geospatial
- Lucene FTS
- MongoDB
- RDF Rank

`ListPatternInterpreter` Interprets a triple pattern where the subject, predicate and context are single values while the **object** is a list of values.

This interface handles triple patterns of this form: `subject predicate (object1 object2 ...) context`.

Open-source plugins using `ListPatternInterpreter`:

- Geospatial

`SubjectListPatternInterpreter` Interprets a triple pattern where the predicate, object and context are single values while the **subject** is a list of values.

This interface handles triple patterns of this form: `(subject1 subject2 ...) predicate object context`.

No open-source plugins use `SubjectListPatternInterpreter` but the usage is similar to `ListPatternInterpreter`.

Pre- and postprocessing handlers

For more information, see *Pre-processing* and *Post-processing*.

`Preprocessor` Allows plugins to maintain a per-query context and have access to `query/getStatements()` properties.

Open-source plugins using `Preprocessor`:

- Lucene FTS
- MongoDB

`Postprocessor` Allows plugins to modify the final result of a query/getStatements() request.

No open-source plugins use `Postprocessor` but the *example plugins* do.

Query request support classes

`Request` A basic read request. Passed to `Preprocess.preprocess()`. Provides access to the `isIncludeInferred` property.

`QueryRequest` An extension of `Request` for SPARQL queries. It provides access to the various constituents of the query such as the `FROM` clauses and the parsed query.

`StatementsRequest` An extension of `Request` for `RepositoryConnection.getStatements()`. It provides access to each of the individual constituents of the request quadruple (subject, predicate, object, and context).

`RequestContext` Plugins may create an instance of this interface in `Preprocess.preprocess()` to keep track of request-global data. The instance will be passed to `PatternInterpreter`, `ListPatternInterpreter`, `SubjectListPatternInterpreter` and `Postprocessor`.

`RequestContextImpl` A default implementation of `RequestContext` that provides a way to keep arbitrary values by key.

Update request handlers

The update request handlers are responsible for processing updates. Unlike the query request handlers, the update handlers are only called for statements that match a predefined pattern.

For more information, see *Update processing*.

`UpdateInterpreter` Handles the addition or removal of statements. Only statements that have one of a set of predefined predicates will be passed to the handler.

The return value determines if the statement will be added or deleted as real data in the repository or processed only by the plugin.

Note that this handler will not be called for each individual statement when removing all statements from all contexts.

Open-source plugins using `UpdateInterpreter`:

- Autocomplete
- GeoSPARQL
- Geospatial
- Lucene FTS
- MongoDB
- Notifications logger
- RDF Rank

`ClearInterpreter` Handles the removal of all statements in a given context or in all contexts.

This handler is especially useful when all statements in all contexts are removed since `UpdateInterpreter` will not be called in this case.

No open-source plugins use `ClearInterpreter`.

`ContextUpdateHandler` Handles the addition or removal of statements in a set of predefined contexts.

You can use this to implement virtual contexts. It is the only part of the Plugin API that does not use integer identifiers but RDF values directly.

No open-source plugins use `ContextUpdateHandler`.

Notification listeners

In general the listeners are used as simple notifications about a certain event such as the beginning of a new transaction or the creation of a new entity.

`EntityListener` Notified about the creation of a new data entity (IRI, blank node, or literal).

Open-source plugins using `EntityListener`:

- Autocomplete

`StatementListener` Notifications about the addition or removal of a statement.

Unlike `UpdateInterpreter`, this listener will be notified about all statements and not just statements with a predefined predicate. The statement will be added or removed regardless of the return value.

Open-source plugins using `StatementListener`:

- Autocomplete
- GeoSPARQL
- Notifications logger

`PluginTransactionListener` and `ParallelTransactionListener` Notifications about the different stages of a transaction (*started*, followed by either *commit* + *completed* or *aborted*).

Plugins should do the bulk of their transaction work within the *commit* stage.

`ParallelTransactionListener` is a marker extension of `PluginTransactionListener` whose *commit* stage is safe to call in parallel with the commit stage of other plugins.

If the plugin does not perform any lengthy operations in the commit stage, it is better to stick to `PluginTransactionListener`.

Open-source plugins using `PluginTransactionListener` or `ParallelTransactionListener`:

- Autocomplete
- GeoSPARQL
- MongoDB
- Notifications logger

Plugin dependencies

For more information, see *Accessing other plugins*.

`PluginDependency` Plugins that need to use other plugins directly must implement this interface. They will be injected into an instance of `PluginLocator`.

`PluginLocator` Provides access to other plugins by name or to the default implementation of `RDFRankProvider`.

`RDFRankProvider` A plugin that provides an RDF rank. The only implementation is the RDF Rank plugin.

Health checks

The health check classes can be used to include a plugin in the repository health check.

`HealthCheckable` Marks a component (a plugin or part of a plugin) as capable of providing health checks. If a plugin implements this interface it will be included in the repository health check.

`HealthResult` The result from a health check. In general health results can be green (everything ok), yellow (needs attention) or red (something broken).

`CompositeHealthResult` A composite health result that aggregates several `HealthResult` instances into a single `HealthResult`.

No open-source plugins implement health checks.

Exceptions

A set of predefined exception classes that can be used by plugins.

`PluginException` Generic plugin exception. Extends `RuntimeException`.

`ClientErrorException` User (client) error, such as invalid input. Extends `PluginException`.

`ServerErrorException` Server error — for example, something unexpected such as lack of disk permissions. Extends `PluginException`.

8.8.4.10 Adding external plugins to GraphDB

With the `graphdb.extra.plugins` property, you can attach a directory with external plugins when starting GraphDB from the command line:

```
graphdb -Dgraphdb.extra.plugins=path/to/directory/with/external/plugins
```

If the property is omitted when starting GraphDB, then you need to load external plugins by placing them in the `dist/lib/plugins` directory and then restarting GraphDB.

Tip: This property is useful in situations when, for example, GraphDB is used in an environment such as Kubernetes where the database cannot be restarted and the `dist` folder cannot be persisted.

8.8.4.11 Putting it all together: example plugins

On Ontotext's github site you can find a project that contains two example plugins, `ExampleBasicPlugin` and `ExamplePlugin`.

ExampleBasicPlugin

`ExampleBasicPlugin` interprets the pattern `?s <http://example.com/now> ?o` and binds the object to a literal containing the system date and time of the machine running GraphDB. The subject position is not used and its value does not matter.

The plugin implements the `PatternInterpreter` interface. It creates a date-time literal as a request-scope entity to avoid cluttering the repository with extra literals.

The plugin extends the `PluginBase` class that provides a default implementation of the `Plugin` interface:

```
public class ExampleBasicPlugin extends PluginBase {
    // The predicate we will be listening for
    private static final String TIME_PREDICATE = "http://example.com/now";

    private IRI predicate; // The predicate IRI
    private long predicateId; // ID of the predicate in the entity pool

    // Service interface methods
    @Override
    public String getName() {
        return "exampleBasic";
    }

    // Plugin interface methods
    @Override
    public void initialize(InitReason reason, PluginConnection pluginConnection) {
        // Create an IRI to represent the predicate
        predicate = SimpleValueFactory.getInstance().createIRI(TIME_PREDICATE);
        // Put the predicate in the entity pool using the SYSTEM scope
        predicateId = pluginConnection.getEntities().put(predicate, Entities.Scope.SYSTEM);

        getLogger().info("ExampleBasic plugin initialized!");
    }
}
```

In this basic implementation, the plugin name is defined, and during initialization, a single system-scope predicate is registered.

Note: It is important to remember to register the plugin in the `META-INF/services/com.ontotext.trree.sdk.Plugin` file in the classpath.

The next step is to implement the first of the plugin's requirements — the pattern interpretation part:

```
public class ExamplePlugin extends PluginBase implements PatternInterpreter {

    // ... initialize() and getName()

    // PatternInterpreter interface methods
    @Override
    public StatementIterator interpret(long subject, long predicate, long object, long context,
                                       PluginConnection pluginConnection, RequestContext
↪requestContext) {
        // Ignore patterns with predicate different than the one we are interested in. We want to return
↪the
```

(continues on next page)

(continued from previous page)

```
        // SystemDate only when we detect the <http://example.com/time> predicate.
        if (predicate != predicateId)
            // This will tell the PluginManager that we cannot interpret the statement so the statement
↪can be passed
            // to another plugin.
            return null;

        // Create the date/time literal. Here it is important to create the literal in the entities
↪instance of the
        // request and NOT in getEntities(). If you create it in the entities instance returned by
↪getEntities() it
        // will not be visible in the current request.
        long literalId = createDateTimeLiteral(pluginConnection.getEntities());

        // return a StatementIterator with a single statement to be iterated. The object of this
↪statement will be the
        // current timestamp.
        return StatementIterator.create(subject, predicate, literalId, 0);
    }

    @Override
    public double estimate(long subject, long predicate, long object, long context,
                           PluginConnection pluginConnection, RequestContext requestContext) {
        // We always return a single statement so we return a constant 1. This value will be used by the
↪QueryOptimizer
        // when crating the execution plan.
        return 1;
    }

    private long createDateTimeLiteral(Entities entities) {
        // Create a literal for the current timestamp.
        Value literal = SimpleValueFactory.getInstance().createLiteral(new Date());

        // Add the literal in the entity pool with REQUEST scope. This will make the literal accessible
↪only for the
        // current Request and will be disposed once the request is completed. Return it's ID.
        return entities.put(literal, Entities.Scope.REQUEST);
    }

}
```

The `interpret()` method only processes patterns with a predicate matching the desired predicate identifier. Further on, it simply creates a new date-time literal in the request scope and places its identifier in the object position of the returned single result. The `estimate()` method always returns 1, because this is the exact size of the result set.

ExamplePlugin

`ExamplePlugin` does the following:

- If a `FROM <http://example.com/time>` clause is detected in the query, the result is a single binding set in which all projected variables are bound to a literal containing the system date and time of the machine running GraphDB.
- If a triple with the subject `http://example.com/time` and one of the predicates `http://example.com/goInFuture` or `http://example.com/goInPast` is inserted, its object is set as a positive or negative offset for all future requests querying the system date and time via the plugin.

The plugin extends the `PluginBase` class that provides a default implementation of the `Plugin` interface:

```
public class ExamplePlugin extends PluginBase implements UpdateInterpreter, Preprocessor, Postprocessor
↪{

    private static final String PREFIX = "http://example.com/";

    private static final String TIME_PREDICATE = PREFIX + "time";
    private static final String GO_FUTURE_PREDICATE = PREFIX + "goInFuture";
    private static final String GO_PAST_PREDICATE = PREFIX + "goInPast";

    private int timeOffsetHrs = 0;

    private IRI timeIri;

    // IDs of the entities in the entity pool
    private long timeID;
    private long goFutureID;
    private long goPastID;


    // Service interface methods
    @Override
    public String getName() {
        return "example";
    }

    // Plugin interface methods
    @Override
    public void initialize(InitReason reason, PluginConnection pluginConnection) {
        // Create IRIs to represent the entities
        timeIri = SimpleValueFactory.getInstance().createIRI(TIME_PREDICATE);
        IRI goFutureIRI = SimpleValueFactory.getInstance().createIRI(GO_FUTURE_PREDICATE);
        IRI goPastIRI = SimpleValueFactory.getInstance().createIRI(GO_PAST_PREDICATE);

        // Put the entities in the entity pool using the SYSTEM scope
        timeID = pluginConnection.getEntities().put(timeIri, Entities.Scope.SYSTEM);
        goFutureID = pluginConnection.getEntities().put(goFutureIRI, Entities.Scope.SYSTEM);
        goPastID = pluginConnection.getEntities().put(goPastIRI, Entities.Scope.SYSTEM);

        getLogger().info("Example plugin initialized!");
    }
}
```

In this implementation, the plugin name is defined, and during initialization, three system-scope predicates are registered.

To implement the first functional requirement the plugin must inspect the query and detect the `FROM` clause in the pre-processing phase. Then, the plugin must hook into the post-processing phase where, if the pre-processing phase detected the desired `FROM` clause, it deletes all query results (in `postprocess()` and returns a single result (in `flush()`) that contains the binding set specified by the requirements. Since this happens as part of pre- and post-processing we can pass the literals without going through the entity pool and using integer IDs.

To do this the plugin must implement `Preprocessor` and `Postprocessor`:

```
public class ExamplePlugin extends PluginBase implements Preprocessor, Postprocessor {
    // ... initialize() and getName()

    // Preprocessor interface methods
    @Override
    public RequestContext preprocess(Request request) {
        // We are interested only in QueryRequests
        if (request instanceof QueryRequest) {
            QueryRequest queryRequest = (QueryRequest) request;
```

(continues on next page)

(continued from previous page)

```
            Dataset dataset = queryRequest.getDataset();

            // Check if the predicate is included in the default graph. This means that we have a "FROM
↪<our_predicate>"
            // clause in the SPARQL query.
            if ((dataset != null && dataset.getDefaultGraphs().contains(timeIri))) {
                // Create a date/time literal
                Value literal = createDateTimeLiteral();

                // Prepare a binding set with all projected variables set to the date/time literal value
                MapBindingSet result = new MapBindingSet();
                for (String bindingName : queryRequest.getTupleExpr().getBindingNames()) {
                    result.addBinding(bindingName, literal);
                }

                // Create a Context object which will be available during the other phases of the request
↪processing
                // and set the created result as an attribute.
                RequestContextImpl context = new RequestContextImpl();
                context.setAttribute("bindings", result);

                return context;
            }
        }
        // If we are not interested in the request there is no need to create a Context.
        return null;
    }

    // Postprocessor interface methods
    @Override
    public boolean shouldPostprocess(RequestContext requestContext) {
        // Postprocess only if we have created RequestContext in the Preprocess phase. Here the
↪requestContext object
        // is the same one that we created in the preprocess(...) method.
        return requestContext != null;
    }

    @Override
    public BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext) {
        // Filter all results. Returning null will remove the binding set from the returned query result.
        // We will add the result we want in the flush() phase.
        return null;
    }

    @Override
    public Iterator<BindingSet> flush(RequestContext requestContext) {
        // Get the BindingSet we created in the Preprocess phase and return it.
        // This will be returned as the query result.
        BindingSet result = (BindingSet) ((RequestContextImpl) requestContext).getAttribute("bindings");
        return new SingletonIterator<>(result);
    }

    private Literal createDateTimeLiteral() {
        // Create a literal for the current timestamp.
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.HOUR, timeOffsetHrs);

        return SimpleValueFactory.getInstance().createLiteral(calendar.getTime());
    }
}
```

The plugin creates an instance of `RequestContext` using the default implementation `RequestContextImpl`. It can

hold attributes of any type referenced by a name. Then the plugin creates a `BindingSet` with the date-time literal, bound to every variable name in the query projection, and sets it as an attribute with the name "bindings". The `postprocess()` method filters out all results if the `requestContext` is non-`null` (that is, if the `FROM` clause was detected by `preprocess()`). Finally, `flush()` returns a singleton iterator that contains the desired binding set in the required case, or it does not return anything.

To implement the second functional requirement that allows setting an offset in the future or the past, the plugin must react to specific update statements. This is achieved by implementing `UpdateInterpreter`:

```
public class ExamplePlugin extends PluginBase implements UpdateInterpreter, Preprocessor, Postprocessor
↪{
    // ... initialize() and getName()

    // ... Pre- and Postprocessor methods

    // UpdateInterpreter interface methods
    @Override
    public long[] getPredicatesToListenFor() {
        // We can filter the tuples we are interested in by their predicate. We are interested only
        // in tuples with have the predicate we are listening for.
        return new long[] {goFutureID, goPastID};
    }

    @Override
    public boolean interpretUpdate(long subject, long predicate, long object, long context, boolean
↪isAddition,
                                   boolean isExplicit, PluginConnection pluginConnection) {
        // Make sure that the subject is the time entity
        if (subject == timeID) {
            final String intString = pluginConnection.getEntities().get(object).stringValue();
            int step;
            try {
                step = Integer.parseInt(intString);
            } catch (NumberFormatException e) {
                // Invalid input, propagate the error to the caller
                throw new ClientErrorException("Invalid integer value: " + intString);
            }

            if (predicate == goFutureID) {
                timeOffsetHrs += step;
            } else if (predicate == goPastID) {
                timeOffsetHrs -= step;
            }

            // We handled the statement.
            // Return true so the statement will not be interpreted by other plugins or inserted in the DB
            return true;
        }

        // Tell the PluginManager that we can not interpret the tuple so further processing can continue.
        return false;
    }
}
```

`UpdateInterpreter` must specify the predicates that the plugin is interested in by using `getPredicatesToListenFor()`. Then, whenever a statement with one of those predicates is inserted or removed, the plugin framework calls `interpretUpdate()`. The plugin then checks if the subject value is `http://example.com/time` and if so handles the update and returns `true` to the plugin framework. This signals that the plugin has processed the update and it doesn't need be inserted as regular data.

8.8.5 Using Maven artifacts

Part of GraphDB's Maven repository is open and allows downloading of GraphDB Maven artifacts without credentials.

Note: You still need to *obtain a GraphDB license*, as the artifacts do not provide one.

8.8.5.1 Public Maven repository

The GraphDB public Maven repository is hosted at `https://maven.ontotext.com/repository/owlim-releases`, and you can browse and search it at https://maven.ontotext.com/#browse/browse:owlim-releases.

You can add the repository to your Maven project's `pom.xml` like this:

```
<repositories>
    <!-- Ontotext's public Maven repository for GraphDB artifacts -->
    <repository>
        <id>graphdb-public</id>
        <url>https://maven.ontotext.com/repository/owlim-releases</url>
    </repository>
</repositories>
```

8.8.5.2 Available artifacts

Note: All Maven snippets below assume you have set the Maven properties `graphdb.version` (and possibly `rdf4j.version`) in the relevant section of your project's `pom.xml` file.

GraphDB distribution

To use the `.zip` distribution for some automation or to run integration tests, download and extract it using the following snippet. The distribution will be unpacked under `target/graphdb-dist`.

```
<build>
   <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.8.1</version>
            <executions>
                <execution>
                    <id>extract-graphdb</id>
                    <phase>package</phase>
                    <goals>
                        <!-- Change this to 'copy' and remove <fileMappers> below,
                             to only download the zip but not extract it -->
                        <goal>unpack</goal>
                    </goals>
                    <configuration>
                        <artifactItems>
                            <artifactItem>
                                <groupId>com.ontotext.graphdb</groupId>
                                <artifactId>graphdb</artifactId>
                                <version>${graphdb.version}</version>
                                <type>zip</type>
```

(continues on next page)

(continued from previous page)

```
                                <classifier>dist</classifier>
                                <outputDirectory>target</outputDirectory>
                                <!-- Removes the version-specific part of the extracted directory -->
                                <fileMappers>
                                    <org.codehaus.plexus.components.io.filemappers.RegExpFileMapper>
                                        <pattern>^\Qgraphdb-${graphdb.version}/\E</pattern>
                                        <replacement>./graphdb-dist/</replacement>
                                    </org.codehaus.plexus.components.io.filemappers.RegExpFileMapper>
                                </fileMappers>
                            </artifactItem>
                        </artifactItems>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

GraphDB Client API library

The GraphDB Client API is a drop-in replacement of RDF4J's `HTTPRepository` class that provides GraphDB extensions, smart GraphDB cluster support, and additional fixes. To use it in a Maven project, add this dependency:

```
<dependency>
    <groupId>com.ontotext.graphdb</groupId>
    <artifactId>graphdb-client-api</artifactId>
    <version>${graphdb.version}</version>
</dependency>
```

GraphDB SDK library

The GraphDB SDK library provides the necessary interfaces to develop GraphDB plugins. To use it in a Maven project, add this dependency:

```
<dependency>
    <groupId>com.ontotext.graphdb</groupId>
    <artifactId>graphdb-sdk</artifactId>
    <version>${graphdb.version}</version>
</dependency>
```

GraphDB runtime library

The GraphDB runtime library provides the core functionality of GraphDB. You may need if you need to embed the database in your application or for unit testing a GraphDB plugin. To embed the database in your application or test a plugin, you need the GraphDB runtime library. To use it in a Maven project, add this dependency:

```
<dependency>
    <groupId>com.ontotext.graphdb</groupId>
    <artifactId>graphdb-runtime</artifactId>
    <version>${graphdb.version}</version>
</dependency>
```

Note: This artifact was previously available from Maven Central but as of GraphDB 11.0.0 this is no longer the case.

8.8.5.3 RDF4J's version

As of GraphDB 11.0.0, the product has migrated to Spring 6, which uses the `jakarta.*` namespace. Since RDF4J has not yet completed its transition from `javax.*` to `jakarta.*`, GraphDB includes a patched version of RDF4J to maintain compatibility.

Since the patched version is not required for most use cases that depend on using GraphDB libraries via Maven, you can safely point your project to stock RDF4J via dependency management like this:

```
<dependencyManagement>
    <!-- We recommend you to put this after other bom-dependencies in your project,
         as RDF4J's bom provides some non-RDF4J versions too. -->
    <dependencies>
        <dependency>
            <groupId>org.eclipse.rdf4j</groupId>
            <artifactId>rdf4j-bom</artifactId>
            <version>${rdf4j.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Note: Jakarta EE is the new name for what was previously known as Java EE (Java Platform, Enterprise Edition) and J2EE (Java 2 Platform, Enterprise Edition). It's a set of specifications and APIs that extend Java SE (Java Standard Edition) with features for enterprise-level applications, like distributed computing and web services. The primary change from Java EE to Jakarta EE was the renaming of the `javax.*` packages to `jakarta.*` to avoid potential trademark issues when the Eclipse Foundation took over management of the Java EE specifications.

8.9 OWL compliance

GraphDB supports several OWL like dialects: OWL-Horst (`owl-horst`), OWL-Max (`owl-max`), which covers most of OWL Lite and RDFS, OWL2 QL (`owl2-ql`), and OWL2 RL (`owl2-rl`).

With the `owl-max` ruleset, GraphDB supports the following semantics:

- full RDFS semantics without constraints or limitations, apart from the entailment related to typed literals (known as D-entailment). For instance, meta-classes (and any arbitrary mixture of class, property, and individual) can be combined with the supported OWL semantics;
- most of OWL Lite;
- all of OWL DLP.

The differences between OWL-Horst and the OWL dialects supported by GraphDB (`owl-horst` and `owl-max`) can be summarised as follows:

- GraphDB does not provide the extended support for typed literals, introduced with the D-entailment extension of the RDFS semantics. Although such support is conceptually clear and easy to implement, it is our understanding that the performance penalty is too high for most applications. You can easily implement the rules defined for this purpose by ter Horst and add them to a custom ruleset;
- There are no inconsistency rules by default;
- A few more OWL primitives are supported by GraphDB (ruleset `owl-max`);
- There is extended support for schema-level (T-Box) reasoning in GraphDB.

Even though the concrete rules pre-defined in GraphDB differ from those defined in OWL-Horst, the complexity and decidability results reported for R-entailment are relevant for TRREE and GraphDB. To be more precise, the

rules in the `owl-horst` ruleset do not introduce new B-Nodes, which means that R-entailment with respect to them takes polynomial time. In KR terms, this means that the `owl-horst` inference within GraphDB is tractable.

Inference using `owl-horst` is of a lesser complexity compared to other formalisms that combine DL formalisms with rules. In addition, it puts no constraints with respect to meta-modeling.

The correctness of the support for OWL semantics (for these primitives that are supported) is checked against the normative Positive- and Negative-entailment OWL test cases.

8.10 W3C specifications

A great advantage of open standards is that the specifications for these standards are freely available. This makes it easier to find official answers about using these standards, as well as learning more about how to better take advantage of each standard. RDF and related standards are produced by the W3C, the same standards body that oversees standards such as HTML and CSS.

8.10.1 Resource Description Framework (RDF)

- RDF Concepts and Abstract Syntax This specification provides a good introduction to basic concepts of RDF (for example, triples, IRIs and URIs, literals, and blank nodes) and the available built-in datatypes that you can use for your data.
- RDF Turtle This spells out all the syntax possibilities for the Turtle syntax used to serialize RDF triples as text files. While the RDF "Concepts and Abstract Syntax" specification mentioned above explains what all the key concepts are, this document shows the specific syntax to encode those concepts and datatypes in your data.
- RDF-star and SPARQL-star Final Community Group Report RDF-star provides a way to let RDF triples themselves serve as the subjects or objects of triples. For example, let's say that:
  - You have a triple stating that `employee38` has a `jobTitle` of 'Assistant Designer'.
  - `employee22` told you this fact about `employee38`.

  RDF-star gives you a way to create a triple with a subject of `employee22`, a predicate of `statedThat`, and an object that consists of the triple (`employee38 jobTitle` 'Assistant Designer')". In other words, the triple about the `employee38` job title is the object of the triple about `employee22`.

  SPARQL-star lets you query for such triples. RDF-star and SPARQL-star improve on the "reification" mechanisms used earlier in RDF's history, which this Community Group Report describes as "verbose and cumbersome." While this Community Group Report is currently the most official W3C document about RDF-star and SPARQL-star, it is not a Recommendation, or official W3C standard. Its contents will be incorporated into the eventual 1.2 updates to the RDF and SPARQL Recommendations. Meanwhile, though, this is a useful document to learn exactly how RDF-star and SPARQL-star work.

8.10.2 SPARQL Protocol and Query Language

Note: See also the GraphDB documentation page about its *SPARQL compliance* and, in the previous section, the "RDF-star and SPARQL-star Final Community Group Report."

- SPARQL Overview For most people, "SPARQL" refers to the query language, but there is actually a set of specification describing the query language and other related capabilities. For example, the JSON, CSV and TSV, and XML query result formats each have their own specification. This document describes a little about each of the SPARQL specifications and where to find out more.
- SPARQL Query Language This is the official specification of all the syntax that is available to create SPARQL `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE` queries. This includes all the functions that these queries

may call; an occasional review of that section in this specification's table of contents can provide a reminder of just how many useful functions are available.

- SPARQL Update This shows all the ways to create an update request (essentially, a specialized query) that inserts and deletes triples in a graph store. This specification also covers operations that you can perform on named graphs such as creating, loading, and dropping them. The *Updating data with SPARQL templates* section of the GraphDB documentation has further information about doing this with data in GraphDB repositories.
- SPARQL Protocol As this specification says, the SPARQL Protocol "describes a means for conveying SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them." GraphDB supports this specification via its *Using the RDF4J REST API's SPARQL endpoints*.
- SPARQL Graph Store HTTP Protocol This specification "describes the use of HTTP operations for the purpose of managing a collection of RDF graphs [as] an alternative to the SPARQL 1.1 Update protocol." GraphDB's support for this specification is described in the *Graph Store HTTP Protocol* section of the documentation.
- SPARQL Federated Query This document shows how to express queries across diverse data sources. GraphDB's support for this specification is described on the *SPARQL Federation* documentation page.

8.10.3 Schemas, ontologies, and data validation

- RDF Schema (Often abbreviated as RDFS) This is one of the original RDF specifications. it describes the specialized classes and properties that let you use RDF triples to create schemas (for example, by defining classes, properties, and the relationships between them) and build on other existing schemas. For more on GraphDB's support of RDFS, see *RDF Schema (RDFS)*.
- OWL 2 Web Ontology Language Document Overview OWL 2 is the most recent version of the set of OWL standards and profiles. This document provides an overview of OWL specifications and where to find out more about each. (See also *The Web Ontology Language (OWL) and its dialects* and *OWL compliance* for more on GraphDB's support of OWL.)
- Shapes Constraint Language (SHACL) The Shapes Constraint Language lets you validate RDF graphs against a set of conditions — for example, to check whether all the instances of a given class have values in their required properties or that certain values are within a specified range. This document shows how to describe all of the different constraints that may be applied. The GraphDB *SHACL validation* page shows how to implement these in a GraphDB repository.

8.11 Benchmarks

Our engineering team invests constant efforts in measuring the database data loading and query answering performance. The section covers common database scenarios tested with popular public benchmarks and their interpretation in the context of common RDF use cases.

8.11.1 LDBC Semantic Publishing Benchmark 2.0

LDBC is an industry association aimed to create TPC-like benchmarks for RDF and graph databases. The association is founded by a consortium of database vendors like Ontotext, OpenLink, Neo Technologies, Oracle, IBM, and SAP, among others. The SPB (Semantic Publishing Benchmark) simulates the database load commonly faced by media or publishing organizations. The synthetic generated dataset is based on BBC's Dynamic Semantic Publishing use case. It contains a graph of linked entities like creative works, persons, documents, products, provenance, and content management system information. All benchmark operations follow a standard authoring process — add new metadata, update the reference knowledge, and search queries hitting various choke points as join performance, data access locality, expression calculation, parallelism, concurrency, and correlated subqueries.

8.11.1.1 Data loading

This section illustrates how quickly GraphDB can do an initial data load. The SPB-256 dataset represents the size of a mid-sized production database managing documents and metadata. The data loading test run measures how the GraphDB edition and the selection of i3 instances affect the processing of 237K explicit statements, including the materialization of the inferred triples generated by the reasoner.

Table 1: Loading time of the LDBC SPB-256 dataset with the default RDFS-Plus-optimized ruleset in minutes

Editions	Ruleset	Explicit statements	Total statements	AWS instance	Cores	Loading time (minutes)
10.4 Free	RDFS-Plus-optimized	256,057,106	401,961,832	r6id.xlarge	1*	247
10.4 SE/EE	RDFS-Plus-optimized	256,057,106	401,961,832	r6id.xlarge	2	232
10.4 SE/EE	RDFS-Plus-optimized	256,057,106	401,961,832	r6id.xlarge	4	230
10.4 SE/EE	RDFS-Plus-optimized	256,057,106	401,961,832	r6id.2xlarge	8	207
10.4 SE/EE	RDFS-Plus-optimized	256,057,106	401,961,832	r6id.4xlarge	16	206

* GraphDB Free uses a single CPU core only.

Loading the dataset with RDF-Plus-optimized ruleset generates an additional nearly 150M implicit statements or expansion of 1:1.6 from the imported explicit triples. GraphDB Free produces the slowest performance due to a limitation of a single write thread. The Standard and Enterprise editions scale with the increase of the available CPU cores until the I/O performance throughput becomes a major limiting factor.

Table 2: Loading time of the LDBC SPB-256 dataset with the default OWL2-RL ruleset in minutes

Editions	Ruleset	Explicit statements	Total statements	AWS instance	Cores	Loading time (minutes)
10.4 SE/EE	OWL2-RL	256,057,106	775,059,522	r6id.xlarge	2	600
10.4 SE/EE	OWL2-RL	256,057,106	775,059,522	r6id.xlarge	4	580
10.4 SE/EE	OWL2-RL	256,057,106	775,059,522	r6id.2xlarge	8	470
10.4 SE/EE	OWL2-RL	256,057,106	775,059,522	r6id.4xlarge	16	486

The same dataset tested with OWL2-RL ruleset produces nearly 515M implicit statements, or an expansion of 1:3.2 from the imported explicit triples. The data loading performance scales much better with the increase of additional CPU cores due to much higher computational complexity. Once again, the I/O performance throughput becomes a major limiting factor, but the conclusion is that datasets with a higher reasoning complexity benefit more from the additional CPU cores.

8.11.1.2 Production load

The test demonstrates the execution speed of small-sized transactions and read queries against the SPB-256 dataset preloaded with RDFS-Plus-optimized ruleset. The query mix includes transactions generating updates and information searches with simple or complex aggregate queries. The different runs compare the database performance according to the number of concurrent read and write clients.

Table 3: The number of executed query mixes per second (higher is better) vs. the number of concurrent clients.

Server instance	Price	Disk	Concurrent read agents	Read query mixes per second	Concurrent write agents	Write per second
r6i.4xlarge	$1.008	EBS (5K IOPS)	0	•	4	22.31
i3.4xlarge	$1.248	local NVMe SSD	0	•	4	21.61
r6id.4xlarge	$1.209	local NVMe SSD	0	•	4	33.63
r6i.4xlarge	$1.008	EBS (5K IOPS)	16	179.75	0	•
i3.4xlarge	$1.248	local NVMe SSD	16	77.58	0	•
r6id.4xlarge	$1.209	local NVMe SSD	16	173.40	0	•
r6i.4xlarge	$1.008	EBS (5K IOPS)	8	115.95	4	17.38
i3.4xlarge	$1.248	local NVMe SSD	8	51.97	4	14.47
r6id.4xlarge	$1.209	local NVMe SSD	8	111.80	4	23.80
r6i.4xlarge	$1.008	EBS (5K IOPS)	12	147.77	4	15.02
i3.4xlarge	$1.248	local NVMe SSD	12	63.70	4	12.10
r6id.4xlarge	$1.209	local NVMe SSD	12	145.20	4	20.14
r6i.4xlarge	$1.008	EBS (5K IOPS)	16	157.07	4	11.24
i3.4xlarge	$1.248	local NVMe SSD	16	70.60	4	8.53
r6id.4xlarge	$1.209	local NVMe SSD	16	158.37	4	15.26

Notes: All runs use the same configuration limited to 20GB heap size on instances with 16 vCPU. The AWS price is based on the US East coast for an on-demand type of instance (Q1 2020), and does not include the EBS volume charges that are substantial only for IOP partitions.

The instances with local NVMe SSD devices substantially outperform any EBS drives due to the lower disk latency and higher bandwidth. In the case of standard and cheapest EBS gp2 volumes, the performance is even slower after the AWS IOPs throttling starts to limit the disk operations. The c5d.4xlarge instances achieve consistently fastest results with the main limitation of small local disks. Next in the list are i3.4xlarge instances offering substantially bigger local disks. Our recommendation is to avoid using the slow EBS volumes, except for cases where you plan to limit the database performance load.

8.11.2 Berlin SPARQL Benchmark (BSBM)

BSBM is a popular benchmark combining read queries with frequent updates. It covers a less demanding use case without reasoning, generally defined as eCommerce, describing relations between products and producers, products and offers, offers and vendors, products and reviews.

The benchmark features two runs, where the "explore" run generates requests like "find products for a given set of generic features", "retrieve basic information about a product for display purpose", "get recent review", etc. The "explore and update" run mixes all read queries with information updates.

Table 4: BSBM 100M query mixes per hour on AWS instance — r6id.4xlarge, local NVMe SSD with GraphDB 10.4 EE, ruleset RDFS-Plus-optimized and excluded Query 5

Threads	explore (query mixes per hour)	explore & update (query mixes per hour)
1	62,170	9,884
2	119,241	13,286
4	217,509	13,320
8	•	13,008
12	•	13,389
16	•	13,307

8.12 GraphDB system statements

System statements are used as SPARQL pragmas specific to GraphDB. They are ways to alter the behavior of SPARQL queries in specific ways. The IDs of system statements are not present in the repository in any way.

GraphDB System Statements can be recognized by their identifiers which begin either with the `onto` or the `sys` prefix. Those stand for `<https://www.ontotext.com/>` and `<http://www.ontotext.com/owlim/system#>`, respectively.

8.12.1 System graphs

System graphs modify the result or change the dataset on which the query operates. The semantics used are identical to standard graphs — the `FROM` keyword. An example of graph usage would be:

```
PREFIX onto: <http://www.ontotext.com/>
SELECT * FROM onto:readwrite WHERE {
        ?s ?p ?o
}
```

System graph	Description
`onto:implicit`	The graph contains statements inferred via the repository's ruleset, located in the default graph.
`onto:explicit`	The graph contains statements inserted in the database by the user, located in the default graph.
`onto:readonly`	The graph contains schematic statements, i.e., the statements which define the repository's ruleset.
`onto:readwrite`	The graph contains non-schematic statements — i.e., all statements beside the ones in the ruleset.
`onto:count`	A pseudo graph that forces a count of the results of the query and returns it as the result.
`onto:disable-sameAs`	A pseudo graph that disables the default behavior of expanding the sameAs nodes of the query result.
`onto:distinct`	A pseudo graph that makes the query behave as if the `DISTINCT` keyword was used.
`onto:skip-redundant-implicit`	A pseudo graph that forces a check if a statement is already explicitly present in the result set and does not return implicit (inferred) versions of the same triple.
`onto:merge`	Specifies that for the given triple pattern merge join should be used. Should be used with a `GRAPH` clause, rather than `FROM`. Merge join is the process of intersecting the current triple pattern with the partial result set which has been already accumulated. They are suitable when there are different collections sharing one or more variables with collection sizes that are relatively equivalent.
`onto:hash`	Specifies that for the given triple pattern hash join should be used. Should be used with a `GRAPH` clause, rather than `FROM`. It performs an intersection between two sets of results by utilizing a hash table.
`onto:explain`	A pseudo graph that returns the query optimization plan.
`onto:commitStatistics`	If enabled, logs commit statistics every 30 seconds.
`sys:statistics`	A pseudo graph that forces the usage of the repository statistics in COUNT queries, instead of properly counting the results when the WHERE clause consists of only one statement pattern. This speeds up the counting operation for simple queries, but it can produce wrong counts where results are manipulated, for example, by `owl:sameAs` expansion.
`onto:retain-bind-position`	Does not allow `BIND` to move freely. Its position is preserved as the original one, relative to the vars that were before it and the ones that were after it.

`onto:describe\/outgoing`	If used, `DESCRIBE` will use forward-only Concise Bounded Description (CBD). The query will return only triples where the described resource is a subject of that triple. It will also recursively follow blank nodes until full closure. This is especially useful for feeding ontology definitions to downstream tools such as Large Language Models (LLMs) for tasks such as SPARQL query generation.
	By default, RDF4J implements symmetric CBD, which includes both outgoing and incoming triples for a resource, which can overwhelm query performance and return irrelevant data.
	A sample `DESCRIBE` query that returns CBD of a resource looks like:
	`DESCRIBE <http://example.com/aReallyGreatBook>`
	`FROM <http://www.ontotext.com/describe/outgoing>`

8.12.2 System predicates

System predicates are used to change the way in which the repository behaves. An example of system predicate usage would be:

```
PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA {
  [] sys:addRuleset "owl-horst-optimized" .
  [] sys:defaultRuleset "owl-horst-optimized" .
  [] sys:reinfer [] .
}
```

System predicate	Description
`sys:schemaTransaction`	Allows for axiom insertion and removal, changing the ruleset.
`sys:reinfer`	Forces full inference re-computation.
`sys:turnInferenceOn`	Enables inferences.
`sys:turnInferenceOff`	Disables inference. This will not remove previously inferred statements.
`sys:addRuleset`	Adds a ruleset.
`sys:removeRuleset`	Removes a ruleset.
`sys:defaultRuleset`	Refers to the default ruleset. Can be used to fetch it or change it.
`sys:currentRuleset`	Refers to the current ruleset. Can be used to fetch it or change it.
`sys:listRulesets`	Lists the currently installed rulesets.
`sys:renameRuleset`	Renames a ruleset.
`sys:exploreRuleset`	Retrieves a ruleset's text, if any.
`sys:consistencyCheckAgainstRuleset`	Checks data consistency against a given ruleset.
`onto:replaceGraph`	Sets a replacement graph. The content of that graph will be replaced with the incoming update. Multiple graphs may be provided by multiple calls with this predicate.
`onto:replaceGraphPrefix`	Sets a prefix for replacement graphs. All graphs whose IRIs start with the prefix will be replaced with the incoming update. Multiple prefixes may be provided by multiple calls with this predicate.
`onto:graphCount`	Returns a count of all named graphs.
`onto:triplesCount`	Returns the total number of triples.
`onto:explicitTriplesCount`	Returns the number of explicit triples.
`onto:implicitTriplesCount`	Returns the number of implicit triples.
`onto:graphs`	Lists all explicit named graphs.

8.13 Glossary

Datalog A query and rule language for deductive databases that syntactically is a subset of Prolog.

D-entailment A vocabulary entailment of an RDF graph that respects the 'meaning' of data types.

Description Logic A family of formal knowledge representation languages that are subsets of first order logic, but have more efficient decision problems.

GraphDB distribution directory The directory where the GraphDB binaries and some additional useful files are stored. See *Distribution directory* for more.

GraphDB home directory The root directory where GraphDB stores all of its data. See *GraphDB home* for more.

Horn Logic Broadly means a system of logic whose semantics can be captured by Horn clauses. A Horn clause has at most one positive literal and allows for an IF…THEN interpretation, hence the common term 'Horn Rule'.

Instance Within the context of clusters in GraphDB, instance is an installation of GraphDB isolated from other GraphDBs installations.

Knowledge Base (In the Semantic Web sense) is a database of both assertions (ground statements) and an inference system for deducing further knowledge based on the structure of the data and a formal vocabulary.

Knowledge Representation An area in artificial intelligence that is concerned with representing knowledge in a formal way such that it permits automated processing (reasoning).

Load Average The load average represents the average system load over a period of time.

Materialization The process of inferring and storing (for later retrieval or use in query answering) every piece of information that can be deduced from a knowledge base's asserted facts and vocabulary.

Named Graph A group of statements identified by a URI. It allows a subset of statements in a repository to be manipulated or processed separately.

Ontology A shared conceptualization of a domain, described using a formal (knowledge) representation language.

OWL A family of W3C knowledge representation languages that can be used to create ontologies. See Web Ontology Language.

OWL-Horst An entailment system built upon RDF Schema, see R-entailment.

Predicate Logic Generic term for symbolic formal systems like first-order logic, second-order logic, etc. Its formulas may contain variables which can be quantified.

RDF Graph Model The interpretation of a collection of RDF triples as a graph, where resources are nodes in the graph and predicates form the arcs between nodes. Therefore one statement leads to one arc between two nodes (subject and object).

RDF Schema A vocabulary description language for RDF with formal semantics.

Resource An element of the RDF model, which represents a thing that can be described, i.e., a unique name to identify an object or a concept.

R-entailment A more general semantics layered on RDFS, where any set of rules (i.e., rules that extend or even modify RDFS) are permitted. Rules are of the form IF…THEN… and use RDF statement patterns in their premises and consequences, with variables allowed in any position.

Resource Description Framework (RDF) A family of W3C specifications for modeling knowledge with a variety of syntaxes.

Semantic Repository A semantic repository is a software component for storing and manipulating RDF data. It is made up of three distinct components:

- An RDF database for storing, retrieving, updating and deleting RDF statements (triples);
- An inference engine that uses rules to infer 'new' knowledge from explicit statements;
- A powerful query engine for accessing the explicit and implicit knowledge.

Semantic Web The concept of attaching machine understandable metadata to all information published on the internet, so that intelligent agents can consume, combine and process information in an automated fashion.

SPARQL The most popular RDF query language.

Statement or Triple A basic unit of information expression in RDF. A triple consists of subject-predicate-object.

Universal Resource Identifier (URI) A string of characters used to (uniquely) identify a resource.

8.14 JavaScript functions

In addition to supporting internal functions such as `NOW()`, `RAND()`, `UUID()`, and `STRUUID()`, GraphDB lets you define and execute your own functions using JavaScript.

JavaScript functions are implemented within the special namespace `<http://www.ontotext.com/js#>`.

8.14.1 Registering JavaScript functions

JavaScript functions are initialized by an `INSERT DATA` request where the subject is a blank node `[]`, `<http://www.ontotext.com/js#register>` is a reserved predicate, and an object of type literal defines your JavaScript code. You can add multiple function definitions at once.

The following example registers two JavaScript functions — `isPalindrome(str)` and `reverse(str)`:

```
prefix extfn:<http://www.ontotext.com/js#>

INSERT DATA {
    [] <http://www.ontotext.com/js#register> '''
        function isPalindrome(str) {
            if (!(str instanceof java.lang.String)) return false;
                rev = reverse(str);
                return str.equals(rev);
        }
        function reverse(str) {
            return str.split("").reverse().join("");
        }
    '''
}
```

This query shows how to retrieve a list of registered JavaScript functions:

```
PREFIX jsfn:<http://www.ontotext.com/js#>
SELECT ?s ?o {
    ?s jsfn:enum ?o
}
```

	s	o

1	isPalindrome	function isPalindrome(str) {
		if (!(str instanceof java.lang.String)) return false;
		var rev = reverse(str);
		return str.equals(rev);
		}

2	reverse	function reverse(str) {
		return str.split("").reverse().join("");
		}

`http://www.ontotext.com/js#enum` is a reserved predicate IRI for listing the available JavaScript functions.

The following example registers a single function to return yesterday's date:

```
PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
    [] jsfn:register '''
    function getDateYesterday() {
        var date = new Date();
        date.setDate(date.getDate() - 1);
        return date.toJSON().slice(0,10);
```

(continues on next page)

(continued from previous page)

```
    }
'''
}
```

We can then use this function in a regular SPARQL query. The following retrieves data created yesterday:

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX jsfn:<http://www.ontotext.com/js#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?s ?date WHERE {
    ?s pubo:creationDate ?date
    FILTER (?date = strdt(jsfn:getDateYesterday(), xsd:date))
}
```

Note: The projected `?date` is filtered by `type` and dynamically assigned `value` — `xsd:date` and the output of the JavaScript function, respectively.

8.14.2 Removing JavaScript functions

De-registering a JavaScript function is similar to registering one. The only difference is the predicate used in the `INSERT` statement: `http://www.ontotext.com/js#remove`.

```
PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
    [] jsfn:remove "getDateYesterday"
}
```

Once removed, the function should be listed as `UNDEFINED`:

	s	o

1	isPalindrome	function isPalindrome(str) {
		if (!(str instanceof java.lang.String)) return false;
		var rev = reverse(str);
		return str.equals(rev);
		}

2	reverse	function reverse(str) {
		return str.split("").reverse().join("");
		}

3	getDateYesterday	UNDEFINED

Note: If multiple function definitions have been registered by a single `INSERT`, removing one of these functions will remove the rest of the functions added by that insert request.

CHAPTER

NINE

HELP

9.1 FAQ

9.1.1 General

9.1.1.1 What is OWLIM?

OWLIM is the former name of GraphDB, which originally came from the term "OWL In Memory" and was fitting for what later became OWLIM-Lite. However, OWLIM-SE used a transactional, index-based file-storage layer where "In Memory" was no longer appropriate. Nevertheless, the name stuck and it was rarely asked where it came from.

9.1.1.2 Why a solid-state drive and not a hard-disk one?

We recommend using enterprise-grade SSDs whenever possible as they provide a significantly faster database performance compared to hard-disk drives.

Unlike relational databases, a semantic database needs to compute the inferred closure for inserted and deleted statements. This involves making highly unpredictable joins using statements anywhere in its indexes. Despite utilizing paging structures as best as possible, a large number of disk seeks can be expected and SSDs perform far better than HDDs in such a task.

9.1.1.3 Is GraphDB Jena-compatible?

Yes, GraphDB provides a standard SPARQL 1.1 endpoint so it is fully interoperable with any SPARQL 1.1 client, including Jena.

9.1.1.4 Can GraphDB run on Redhat or Centos?

Yes. GraphDB can run on any Linux distribution.

9.1.1.5 Can GraphDB load archived files?

GraphDB supports importing gzip and zip compressions.

9.1.1.6 Can I have a GraphDB Free Docker image?

Yes, but this is something you have to do manually, since GraphDB Free requires a request form. If you don't have Makefile, you can run the docker build command directly:

```
docker build --no-cache --pull --build-arg edition=free --build-arg version=${VERSION} -t
↪ontotext/graphdb:${VERSION}-free free-edition
```

9.1.1.7 Is there a Python/Java/C#/Javascript/REST API?

- Java has the official RDF4J and *GraphDB API*.
- JavaScript has the official GraphDB client.
- Python has the official rdflib (since 7.5.0) client.
- C# has dotNetRdf.
- The REST API is available from swagger and documented at RDF4J.

9.1.2 Configuration

9.1.2.1 How do I find out the exact version number of GraphDB?

The major/minor version and patch number are part of the GraphDB distribution `.zip` file name. They can also be seen at the bottom of the GraphDB Workbench home page, together with the RDF4J, Connectors, and Plugin API's versions.

A second option is to run the **`graphdb -v`** startup script command if you are running *GraphDB as a standalone server* (without Workbench). It will also return the build number of the distribution.

Another option is to run the following `DESCRIBE` query in the Workbench SPARQL editor:

```
DESCRIBE <http://www.ontotext.com/SYSINFO> FROM <http://www.ontotext.com/SYSINFO>
```

It returns pseudo-triples providing information on various GraphDB states, including the number of triples (total and explicit), storage space (used and free), commits (total and whether there are any active ones), the repository signature, and the build number of the software.

9.1.2.2 What is a location?

A location is either a local (to the Workbench installation) directory where your repositories will be stored or a remote instance of GraphDB. You can have multiple attached locations but only a single location can be active at a given time.

9.1.2.3 How do I attach a location?

Go to *Setup* ▸ *Repositories*. Click *Attach remote location*. For a location on the same machine, provide the absolute path name to a directory, and for a remote location, provide a URL through which the server running the Workbench can see the remote GraphDB instance.

9.1.2.4 Does GraphDB do backups on its own?

No, you have to automate the backup. GraphDB has full and incremental backup options. A cron job is reliable and easy to set up.

9.1.3 RDF and SPARQL

9.1.3.1 How is GraphDB related to RDF4J?

GraphDB is a semantic repository, packaged as a Storage and Inference Layer (Sail) for the RDF4J framework and it makes extensive use of the features and infrastructure of RDF4J, especially the RDF model, RDF parsers, and query engines.
For more details, see the GraphDB *RDF4J*.

9.1.3.2 What does it mean when an IRI starts with `urn:rdf4j:triple:`?

When RDF-star (formerly RDF*) embedded triples are serialized in formats (both RDF and query results) that do not support RDF-star, they are serialized as special IRIs starting with `urn:rdf4j:triple:` followed by Base64 URL-safe encoding of the N-Triples serialization of the triple. This is controlled by a boolean writer setting, and is ON by default. The setting is ignored by writers that support RDF-star natively.

Such special IRIs are converted back to triples on parsing. This is controlled by a boolean parser setting, and is ON by default. It is respected by all parsers, including those with native RDF-star support.

See *RDF-star and SPARQL-star*.

9.1.3.3 What kind of SPARQL compliance is supported?

All GraphDB editions support:

- SPARQL 1.1 Protocol for RDF
- SPARQL 1.1 Query
- SPARQL 1.1 Update
- SPARQL 1.1 Federation
- SPARQL 1.1 Graph Store HTTP Protocol

See also *SPARQL compliance*.

9.1.3.4 Can I use SHACL with a specific context only?

No. Using SHACL with a specific context only is outside of the scope for standard SHACL. In the future, we may have a SHACL specification extension that covers this functionality.

9.1.3.5 Can I have partial inserts with SHACL?

No. Partial inserts support — such as blocking only the problematic data and letting the "good" data in — is unlikely to ever be implemented. If you allow partial insert, you limit traceability. There's also the issue that your data may be recursively dependent, resulting in the need for extra checks.

9.1.3.6 Can I get performance information about my queries?

Press Shift-[Ctrl|Cmd]-Enter to get the explain plan, and see how the query would get evaluated.

9.1.3.7 Can I create groups of saved queries?

No. If you want different users to have access to different queries, you can do it by creating separate users.

9.1.3.8 Can I export a single named graph?

Yes. This can be done through the workbench UI or through the RDF4J's REST API.

9.1.4 Security

9.1.4.1 Does GraphDB have any security vulnerabilities?

Every software potentially exposes security vulnerabilities, mainly when it depends on several third-party libraries like Spring, Apache Tomcat, JavaScript frameworks, etc. The GraphDB team does everything possible to constantly fix and discover new vulnerabilities using OWASP dependency check, Trivy, and Snyk packages. In addition, every GraphDB release is checked for any publicly known vulnerabilities and all suspected issues with score High are investigated.

9.1.4.2 Does the Log4Shell issue (CVE-2021-44228) affect GraphDB?

No, it is not affected. All GraphDB editions and plugins between 6.x and 9.x use Logback, but not Apache Log4j 2; thus, our users are safe in terms of CVE-2021-44228 (aka Log4Shell).

9.1.5 Troubleshooting

9.1.5.1 Why can't I use custom rule file (`.pie`) — an exception occurred?

To use custom rule files, GraphDB must be running in a JVM that has access to the Java compiler. The easiest way to do this is to use the Java runtime from a Java Development Kit (JDK).

9.1.5.2 Why can't I open GraphDB in MacOS?

If you receive an error message saying that MacOS cannot open GraphDB since it cannot be checked for malicious software, this is because the security settings of your Mac are configured to only allow apps from the App Store.

GraphDB is a developer-signed software, so in order to install it, you need to modify these settings to allow apps from both the App Store and identified developers.

You can find detailed assistance on how to configure them in the Apple support pages.

9.1.5.3 How to workaround "Insufficient disk space to start a transaction for repository" error?

GraphDB prevents you from starting transactions while you have critically low disk space. To disable this protection check, set both `graphdb.health.minimal.free.storage.enabled` and `graphdb.health.minimal.free.storage.asyncCheck` to `false`. Check the *low disk space parameters* documentation for more information.

9.1.5.4 The login form does not show up, security is on, and I cannot log in. What to do?

The login form is set to appear when `graphdb.auth.methods` is set to *"gdb"* or *"openid"*, or both. If you only have *"basic"*, this means that the database expects you to pass your username and password in the URL.

9.1.5.5 Why is the database storage not getting smaller when I delete triples?

Deleted data is kept inside the database, but marked as unavailable, so if you reload it, it would be fast. You can compact the indexes by using the *rebuild option of the storage tool*.

9.2 Support

- email: graphdb-support@ontotext.com
- Twitter: @OntotextGraphDB
- GraphDB tag on Stack Overflow at http://stackoverflow.com/questions/tagged/graphdb

9.3 Release notes

GraphDB release notes provide information about the features and improvements in each release, as well as various bug fixes. GraphDB's versioning scheme is based on semantic versioning. The full version is composed of three components:

`major.minor.patch`

e.g., `11.2.0` where the major version is `11`, the minor version is `2` and the patch version is `0`.

Occasional versions may include a modifier after a hyphen, e.g., `11.2.0-RC1` to signal additional information, for example, a test release (TR1, TR2 and so on), a release candidate (RC1, RC2 and so on), a milestone release (M1, M2 and so on), or other relevant information.

Note: Releases with the same major and minor versions do not contain any new features. Releases with different patch versions contain fixes for bugs discovered since the previous minor version. New or significantly changed features are released with a higher major or minor version.

GraphDB 11.3 includes the following components with their version numbers:

- RDF4J
- GraphDB Connectors
- GraphDB Workbench

Their versions use the same semantic versioning scheme as the whole product, and their values are provided only as a reference.

9.3.1 GraphDB 11.3.3

Released: 23 April 2026

9.3.1.1 Component versions

RDF4J	Connectors	Workbench
5.2.2	16.6.2	3.3.3

Important: GraphDB 11.3.3 addresses several external vulnerabilities, including a critical vulnerability affecting the Online Certificate Status Protocol. This release also fixes several bugs, such as one which causes unwanted snapshot replication, and one where queries for compacting the indexes are not forwarded.

We recommend everyone to upgrade.

9.3.1.2 Core database

Bug fixing

	Description	Introduced	Severity
GDB-14373	Working with triples in RDF-star may result in an error	11.3.1	Major
GDB-14311	Cannot compact indexes on SHACL repositories	9.x	Major
GDB-13626	GraphDB startup fails during Java version detection when the environment variable `_JAVA_OPTIONS` is set	11.2	Medium

9.3.1.3 Cluster

Bug fixing

	Description	Introduced	Severity
GDB-14404	Cluster formation and communication fails when the x509 certificate authentication is configured using the new `graphdb.connector.ssl.*` namespace	11.2	Major
GDB-14323	Trying to save a query without a name causes snapshot replication	Major	

9.3.1.4 Security

Fixed third-party vulnerabilities

Vulnerability	Severity
CVE-2026-22735	2.6 Low
CVE-2026-22737	5.9 Medium
CVE-2026-24880	7.5 High
CVE-2026-25854	6.1 Medium
CVE-2026-29129	7.5 High
CVE-2026-29145	9.1 Critical
CVE-2026-29146	7.5 High
CVE-2026-32990	5.3 Medium
CVE2026-34237	6.1 Medium
CVE-2026-34483	7.5 High
CVE-2026-34487	7.5 High
CVE-2026-34500	6.5 Medium
CVE-2026-35554	8.7 High
CVE-2026-35568	5.1 Medium

See also:

Third-party vulnerabilities for all other known high or critical third-party vulnerabilities.

9.3.2 GraphDB 11.3.2

Released: 07 April 2026

9.3.2.1 Component versions

RDF4J	Connectors	Workbench
5.2.2	16.6.1	3.3.3

Important: GraphDB 11.3.2 fixes several critical security issues and third-party vulnerabilities. Notably, this release addresses a critical vulnerability where XML parsers could be used as a potential vector for an XXE attack — to resolve this issue, retrieving XML files that contain `DOCTYPE` declarations is no longer supported when performing federated SPARQL queries, in accordance to the specification. Another critical vulnerability where an RCE attack could be potentially carried out through the Avatica translation layer has also been resolved. This release also addresses a number of major or less severe bugs related to the core functionalities of GraphDB, the cluster mechanism, GraphQL and the GraphDB Workbench, among other things.

We recommend everyone to upgrade.

9.3.2.2 Core database

Bug fixing

	Description	Introduced	Severity
GDB-14149	Nested optional filter assigns non-existent values (`BINDs`) instead of filtering	11.3	Major
GDB-13665	Transaction rollbacks of concurrent updates is not always isolated	11.2	Medium

9.3.2.3 Cluster

Bug fixing

	Description	Introduced	Severity
GDB-14283	Leader didn't attempt to catch up node after receiving higher term and neither step down	10.0	Major
GDB-13720	Invalid repo configuration breaks the cluster	Major	11.2

9.3.2.4 Security

New features and improvements

- GDB-14300 "Invalid direct role" LDAP warning log has been demoted to debug level

Bug fixing

	Description	Introduced	Severity
GDB-14334	Vulnerability with potential XXE attacks through XML parsers — Disallows DOCTYPE **Note:** When performing federated SPARQL queries, ensure that DTD (Document Type Definition)/`DOCTYPE` declarations are excluded, as they are no longer supported.		Critical
GDB-14326	x509 certificates are incorrectly decoded from HTTP headers	10.7	Major
GDB-14251	Remote Code Execution via Avatica translation layer		Critical
GDB-14240	Users with GRAPHQL-only rights defined in the LDAP can still access the SPARQL `READ` functionality via the REST API	11.3	Major
GDB-13749	Local Path traversal in repository ID allows file writes to arbitrary locations on disk		Medium

Fixed third-party vulnerabilities

Vulnerability	Severity
CVE-2026-24734	7.5 High
CVE-2026-33871	8.7 High
CVE-2026-33870	7.5 High
CVE-2026-22732	9.1 Critical

9.3.2.5 GraphQL

Bug fixing

	Description	Introduced	Severity
GDB-14304	GraphQL relations attribute counting counts the relations themselves in the results count when there are inverse or symmetric relations	11.0	Major
GDB-14290	Users can see the count of all sub-entities even when lacking the permission to view the sub-entities themselves		Major
GDB-13541	"Cannot rollback transaction already committed transaction for request" randomly returned in Semantic Objects	11.2	Medium

9.3.2.6 Workbench & UX

New features and improvements

- GDB-14221 Allow overriding the Base URL protocol for the Swagger UI

Bug fixing

	Description	Introduced	Severity
GDB-14241	Pressing the SPARQL redirect button under SPARQL Explain in Talk to Your Graph does not add the necessary prefixes in the editor	11.0	Medium
GDB-14239	SPARQL Editor buttons are hidden behind the scrollbar if the browser has been zoomed out	11.3.1	Minor
GDB-14236	Pressing *Cancel* when in the *Users and Access* ▸ *Edit user permissions* menu does not revert back to user list	11.0	Minor
GDB-14183	Error occurs when pressing *Save* button in the *Edit/Clone Agent* dialog in Talk to Your Graph		Medium
GDB-14166	Loading the visual graph from the SPARQL view takes too long	11.2.1	Medium
GDB-13846	Resizing the Welcome page stacks the banner text	11.3	Minor
GDB-13806	Navbar is rendered on login page after GDB instance is restarted	11.3	Minor
GDB-6208	Switching to a FedX repository from a worker repository in a cluster when in `/namespaces` view caches the namespaces list	9.10	Medium

9.3.3 GraphDB 11.3.1

Released: 19 March 2026

9.3.3.1 Component versions

RDF4J	Connectors	Workbench
5.2.2	16.6.1	3.3.2

Important: GraphDB 11.3.1 fixes a number of major bugs. This update resolves a bug introduced in GraphDB 11.3 where creating and editing users through the Workbench in GraphDB 11.3 may cause a critical malfunction for features using custom user roles. Two major bugs affecting working with the cluster were also addressed, one which caused GraphDB to stop responding during a partial restore, and another where unreachable nodes in a cluster deployment could not be removed through the GraphDB Workbench. Another two major bugs which caused SPARQL to return incorrect result — an error when processing `Extension/BIND` within `FILTER EXISTS`, and the `DISTINCT SPARQL` not working as expected — are also fixed in this release. Finally, two major security issues and a number of third-party vulnerabilities are also addressed with GraphDB 11.3.1.

In addition to these major bugs, GraphDB 11.3.1 fixes a large number of other issues with the GraphDB Workbench, most of which were introduced with GraphDB 11.3.

We recommend everyone to update.

9.3.3.2 Core database

Bug fixing

	Description	Introduced	Severity
GDB-14219	Processing `Extension/BIND` statements within `FILTER/EXISTS` causes an invalid query evaluation	11.1.1	Major
GDB-13850	The `DISTINCT SPARQL` operator does not work	10.8.8	Major
GDB-13748	Calls for internal federation force all `config.ttl` files to be read every time, affecting performance	11.0.2	Medium
GDB-13078	SPARQL `FILTER` is ignored when using concat in the same optional block	10.8.6	Medium

9.3.3.3 Cluster

Bug fixing

	Description	Introduced	Severity
GDB-13784	Cannot edit a cluster in the GraphDB Workbench when there are unreachable nodes		Major
GDB-13384	GraphDB stops responding during a partial restore		Major

9.3.3.4 Security

Bug fixing

	Description	Introduced	Severity
GDB-13857	SecurityManagementController returns encrypted passwords	9.x	Major
GDB-13745	Path traversal vulnerability leads to data exposure and potential file system bruteforce attacks		Major

Fixed third-party vulnerabilities

Vulnerability	Severity
CVE-2026-24281	7.4 High
CVE-2026-24308	7.5 High
GHSA-72hv-8253-57qq	8.7 High
CVE-2026-24733	3.7 Low
CVE-2025-66614	9.1 Critical
CVE-2025-67735	6.5 Medium

9.3.3.5 GraphQL

Bug fixing

	Description	Introduced	Severity
GDB-13772	Bad SPARQL template parsing when creating a SPARQL query	11.0	Major

9.3.3.6 LLM & AI

Bug fixing

	Description	Introduced	Severity
GDB-14150	TTYG cannot load users if OAuth or LDAP is configured, but security is disabled	10.8	Medium
GDB-13781	The vector is not resolved when executing a vector query with highlight	11.2	Medium

9.3.3.7 Workbench & UX

Bug fixing

	Description	Introduced	Severity
GDB-14212	Operation notifications are not showin in the header	11.3	Medium
GDB-14210	Query Graph on System Monitoring is always on 0	11.2	Medium
GDB-14186	*Open in SPARQL* button in TTYG incorrectly redirects to root context when GraphDB is behind a reverse proxy	11.2	Minor
GDB-14148	The size of import files is displayed in millibits (mb) instead of megabytes (MB)	11.3	Minor
GDB-14144	Import error messages are hard to read due to poor contrast	11.3	Medium
GDB-14129	Visual graph label length causes overlap between labels	11.3	Medium
GDB-14120	Cookie consent banner is visible in login page	11.2	Medium
GDB-14118	Agent settings textbox is cutoff when deleiting a TTYG agent in a non-maximized window	11.2.1	Medium
GDB-14115	User-assigned ACL roles incorrectly prepend with `CUSTOM` after upgrading to 11.3.0	11.3	Major
GDB-14102	Visual Graph's text labels are not centered on Safari browsers	11.3	Medium
GDB-14101	Zooming in or out in Visual Graph does not work on Safari	11.3	Medium
GDB-14100	Icons missing from login form when using the dark theme	11.3	Medium
GDB-14094	Various tooltip inconsistencies after rebranding the Workbench	11.3	Medium
GDB-13869	GraphQL Playground loses some styling elements after refreshing the page	11.3	Minor
GDB-13851	The language tooltip label does not update after changing the language	11.0	Minor
GDB-13844	Creating SPARQL templates with `#` in the IRI adds the repository ID and breaks the view	11.0	Medium
GDB-13843	Duplicated repository ID context URL upon login with reverse proxy, which results in a 404 error	11.3	Medium
GDB-13842	Missing repository ID URL context parameter after logging in, then logging out	11.3	Medium
GDB-13774	SPARQL view *Keyboard shortcuts* button is cutoff when there is a scroll bar in the editor	11.3	Medium
GDB-13773	The languages tooltip blocks the repository dropdown menu	11.3	Minor
GDB-13754	Welcome View banner through external proxy REST API redirect is not working	11.0	Minor
GDB-13706	"Unable to load Remix icon" console spam on Firefox	11.3	Minor
GDB-13655	Opening a Talk To Your Graph new chat randomly loses styling	11.3	Minor
GDB-12548	Empty prefix is not automatically added	11.1	Medium

9.3.4 GraphDB 11.3.0

Released: 19 February 2026

9.3.4.1 Component versions

RDF4J	Connectors	Workbench
5.2.2	16.6.0	3.3.1

Important: GraphDB 11.3 introduces major new features and critical bug fixes, including a significant update to the look and usability of the GraphDB Workbench. Key updates include backup validation to confirm when a backup has been successful, support for multiple MCP versions, the ability for URL links to change the active repository in the GraphDB Workbench, and further improvements to the new native Python client introduced in GraphDB 11.2. To improve stability and performance, this release also fixes several major bugs affecting the SPARQL editor, as well as compliance issues when subtracting two dates, which should now return `xs:dayTimeDuration` as prescribed by the SPARQL specification.

Additionally, GraphDB 11.3 includes numerous other bug fixes and smaller improvements, enhancing overall product stability and user experience. The versions of several external components and connectors have also been updated, and some migration steps may be necessary to ensure your GraphDB deployment is working properly after upgrading to 11.3 — check the *Migrating GraphDB configurations page* for more details.

The development of the features in this release was partially funded by the EnrichMyData (GA nu: 101070284) Horizon Europe research project.

A more accessible Workbench with a fresh new look The GraphDB Workbench has been updated to reflect our rebranding to Graphwise with new brand colors and a cleaner, more intuitive design. While the layout and functionality of the Workbench largely remain the same, it is now possible to change the active repository in GraphDB by loading a link, making collaborative work between different GraphDB instances easier. Lastly, we have also expanded the list of officially supported browsers, which now includes Safari and Microsoft Edge.

Improved backup fetching GraphDB now informs you whether your backup has been successfully processed. The engine now also includes a mechanism that prevents attempts to apply a corrupt or incomplete backup.

Enhanced MCP support GraphDB 11.3 significantly expands its Model Context Protocol (MCP) capabilities by adding support for multiple MCP versions, including the latest 2025-11-25 standard. Developers now also have the option to choose between legacy SSE and the new Streamable HTTP transport. This flexibility allows you to optimize for high-concurrency environments and handle resource exhaustion.

This change also breaks any existing MCP configuration. Check the *Migrating GraphDB configurations page* for more details.

Advanced vector search in Elasticsearch and OpenSearch You can now create vector fields in nested objects in the *Elasticsearch* and *OpenSearch* connectors for more nuanced and complex similarity searches within your knowledge graph.

Warning: A bug in GraphDB 11.3 may affect deployments which use the *Custom user roles* mechanism

Creating and editing users through the Workbench in GraphDB 11.3 may cause a critical malfunction for features using custom user roles, such as *Access control list* in *Fine-grained access control* and *Role-Based Access Control* in GraphQL. When a `CUSTOM` role is defined for a user, editing that user through the Workbench will incorrectly save the group with an additional `CUSTOM_` prefix. This will happen when any of that user's attributes were modified and saved, even if the `CUSTOM` role itself was not changed.

An additional `CUSTOM_` prefix is also erroneously shown in the users list view in the Workbench; however, if the user configurations have not been edited, it will not affect any other functionalities.

> This bug is present only when editing users through the GraphDB Workbench. If you need to edit a user's configurations in GraphDB 11.3.0, do not use the Workbench. You can safely use the GraphDB REST API instead, as it will not cause any issues.
>
> If you encounter this bug, you can fix the issue by removing the additional `CUSTOM_` prefix from the group through the REST API.
>
> This bug will be fixed in GraphDB 11.3.1 which will be released as soon as possible.

9.3.4.2 Core database

New features and improvements

- GDB-13563 Add support for the Ontop configuration keys that are not yet supported in the GraphDB Ontop integration
- GDB-13547 Introduce a mechanism to ensure a backup is successful before applying it
- GDB-13289 Add system predicates which fetch graphs names and statement counts

Bug fixing

	Description	Introduced	Severity
GDB-13674	When using a GraphDB Free license and the maximum number of repositories is reached, attempting to edit a repository results in a "No valid license" error	11.0	Medium
GDB-13639	Running an "Explain query plan" from the SPARQL editor button results in an error	11.2.1	Major
GDB-13606	Subtraction of two dates is incompliant with the SPARQL specification and returns `xs:duration` instead of `xs:dayTimeDuration`	9.x	Major
GDB-12390	Renaming a repository to the same name as an existing repository returns the wrong error message		Minor
GDB-5053	The GraphDB Optimizer incorrectly evaluates functions used inside `BIND/Extension` parts of nested queries once	9.x	Medium

9.3.4.3 Security

New features and improvements

- GDB-13523 Add support for LDAP group prefix/suffix

Fixed third-party vulnerabilities

Vulnerability	Severity
CVE-2026-1225	1.8 Low

9.3.4.4 GraphQL

Bug fixing

	Description	Introduced	Severity
GDB-13690	PropertyRangeValidator fails to validate property range covariant if the range type does not exist	11.0	Medium

9.3.4.5 LLM & AI

New features and improvements

- GDB-13582 Add `WebMvcStreamableServerTransportProvider` and update the default MCP transport configuration
- GDB-13559 Upgrade the distributed MCP server to the latest version of the protocol MCP version 2025-11-25
- GDB-13356 Update and optimize the *Talk to your graph* UI with a new response for similarity indexes

Bug fixing

	Description	Introduced	Severity
GDB-13178	After cancelling a prompt, the explain button on the previous prompt stops working	11.2	Minor

9.3.4.6 External integrations

New features and improvements

- GDB-13263 Display a more meaningful message when attempting to perform a similarity query on a non-vector field in Elasticsearch
- GDB-12072 Improve error message when max clause count is reached in Lucene

Bug fixing

	Description	Introduced	Severity
GDB-13486	Cannot create a vector field from a nested object field in Elasticsearch and OpenSearch	11.2	Medium
GDB-13327	`graphdb.opensearch.connector.knn_k` returns an inconsistent number of results with simple OpenSearch vector queries	11.2	Medium

9.3.4.7 Workbench & UX

New features and improvements

- GDB-13302 Redesign the GraphDB Workbench to reflect our rebranding to Graphwise
- GDB-12488 Add a copy button for the Explain plan of a SPARQL query
- GDB-5601 Introduce a repository URL parameter that changes the active repository in GraphDB when the link is opened in Workbench
- GDB-5236 Truncate long node labels in the Visual Graph view

Bug fixing

	Description	Introduced	Severity
GDB-13763	Saved queries cannot be shared with users	11.2.1	Medium
GDB-13729	Untranslated message in the GraphDB Workbench JDBC during query preview	11.2	Minor
GDB-13662	Status bar is displayed in black and white when using Workbench in Firefox		Medium
GDB-13646	REST API with dark mode is hard to read when logging in on the view	11.2	Minor
GDB-13644	Class relationship view sometimes renders multiple diagrams on top of each other		Medium
GDB-13625	Repository selector icon disappears when the Workbench is loaded in multiple tabs		Minor
GDB-13548	Expansion ratio is returned without decimal values	11.1	Medium
GDB-13542	The login button is displaced when OpenID Connect and Basic Authentication are both enabled	11.2.1	Minor
GDB-13493	Embedded Visual Graph does not work	10.7	Medium
GDB-13473	The Resource filter disappears when the browser window is resized		Minor
GDB-13409	When the Workbench is loaded in two separate tabs, changing the active repository in one tab results in an error in the other	11.1	Medium
GDB-13165	Unwanted requests to the backend when permission banner is shown	11.2	Medium

9.4 Third-party vulnerabilities

GraphDB uses a number of third-party libraries as dependencies. To ensure a secure and reliable product, Ontotext performs regular vulnerability scans and takes appropriate actions to address any newly identified vulnerabilities.

Typically, vulnerabilities are addressed by upgrading the affected libraries to a newer version, where the vulnerability was fixed.

When it is not possible to upgrade a library, Ontotext addresses each case individually. This may include a statement that a vulnerability does not apply to GraphDB, an Ontotext patch for that vulnerability, or both.

Currently, there is one high vulnerability, and there are no known identified critical vulnerabilities in GraphDB 11.3.3:

Vulnerability	Severity	Description
CVE-2026-2332	7.4 High	CVE-2026-2332 is an HTTP/1.1 request smuggling vulnerability in Jetty's server-side `HttpParser`. Our usage of `jetty-http` is exclusively as an outbound HTTP client. The vulnerable server-side parsing code path is never executed in our deployment. The attack preconditions are not met. Therefore this is not exploitable in this configuration.