GraphDB Documentation

Release 10.0.0

Ontotext

23 August 2022
## CONTENTS

1 General | 1
---|---
1.1 About GraphDB | 2
  1.1.1 Comparison of GraphDB Free, GraphDB Standard, and GraphDB Enterprise | 3
1.2 Architecture & Components | 3
  1.2.1 Architecture | 4
    1.2.1.1 RDF4J | 5
    1.2.1.2 The Sail API | 5
  1.2.2 Components | 5
    1.2.2.1 Engine | 6
    1.2.2.2 Connectors | 6
    1.2.2.3 Workbench | 6
1.3 Cluster Basics | 6
  1.3.1 High availability features | 7
    1.3.1.1 Quorum-based replication | 7
    1.3.1.2 Internal and external proxy | 7
    1.3.1.3 Query load balancer | 7
  1.3.2 Cluster roles | 7
  1.3.3 Terms | 8
  1.3.4 Log replication | 8
  1.3.5 Leader election | 9
1.4 Connectors | 10
  1.4.1 Full-text search and aggregation connectors | 10
  1.4.2 MongoDB integration | 11
  1.4.3 Kafka connector | 11
1.5 Workbench | 11
1.6 Benchmarks | 12
  1.6.1 LDBC Semantic Publishing Benchmark 2.0 | 12
    1.6.1.1 Data loading | 13
    1.6.1.2 Production load | 14
  1.6.2 Berlin SPARQL Benchmark (BSBM) | 15

2 Quick Start Guide | 17
  2.1 Run GraphDB as a Desktop Installation | 17
    2.1.1 On Windows | 17
    2.1.2 On MacOS | 17
    2.1.3 On Linux | 18
    2.1.4 Configuring GraphDB | 18
    2.1.5 Configuring the JVM | 19
    2.1.6 Stopping GraphDB | 19
  2.2 Run GraphDB as a Standalone Server | 19
    2.2.1 Running GraphDB | 20
      2.2.1.1 Options | 20
    2.2.2 Configuring GraphDB | 20
      2.2.2.1 Paths and network settings | 20
3.7 Using Maven Artifacts

3.6 Distribution Package

3.5 GraphDB Command Line Tools

3.4 Migrating GraphDB Configurations

3.3 Configuring GraphDB

3.2 Running GraphDB

3.1 Requirements

2.8 Additional Resources

2.7 Query Your Data

2.6 Explore Your Data and Class Relationships

2.5 Load Your Data

2.4 Create a Repository

2.3 Set up Your License

2.2.2 Java Virtual Machine settings

2.2.1 Stopping the database

2.2.2 Load data through the GraphDB Workbench

2.2.3 Load data through SPARQL or RDF4J API

2.2.4 Load data through the ImportRDF tool

2.2.1 Explore instances

2.2.2 Create your own visual graph

2.2.3 Class hierarchy

2.2.4 Domain-Range graph

2.2.5 Class relationships

2.1 Requirements

2.1.3 Memory management

2.1.4 Licensing

2.1.5 Hardwaresizing

2.1.6 Minimum requirements

2.1.7 Licensing

2.1.8 Memory management

2.1.9 Hardwaresizing

2.1.10 Minimum requirements

2.1.11 Licensing

2.1.12 Memory management

2.1.13 Hardwaresizing

2.1.14 Minimum requirements

2.1.15 Licensing

2.1.16 Memory management

2.1.17 Hardwaresizing

2.1.18 Minimum requirements

2.1.19 Licensing

2.1.20 Memory management

2.1.21 Hardwaresizing

2.1.22 Minimum requirements

2.1.23 Licensing

2.1.24 Memory management

2.1.25 Hardwaresizing

2.1.26 Minimum requirements
4 Administration

4.1 Workbench User Interface .............................................. 63
  4.1.1 Functionalities .................................................. 64
  4.1.2 Configuration properties ....................................... 66

4.2 Setting up Licenses ..................................................... 66
  4.2.1 Setting up licenses through the Workbench .................. 67
  4.2.2 Setting up licenses through a file ............................ 68
    4.2.2.1 Custom file path property ................................. 68
  4.2.3 Order of preference ............................................ 68

4.3 Creating Locations ................................................... 68
  4.3.1 Connect to a remote location .................................. 69
    4.3.1.1 Change location settings ................................. 70
    4.3.1.2 View or update location license ......................... 71
  4.3.2 Configure a data location ..................................... 71

4.4 Creating a Repository ................................................ 71
  4.4.1 Create a repository ............................................. 71
    4.4.1.1 Using the Workbench ...................................... 71
    4.4.1.2 Using the RDF4J console ................................. 73
  4.4.2 Manage repositories ............................................ 73
    4.4.2.1 Connect a repository ..................................... 73
    4.4.2.2 Make it the default repository ........................... 74
    4.4.2.3 Edit a repository ......................................... 74

4.5 Configuring a Repository ............................................ 74
  4.5.1 Plan a repository configuration ................................ 75
  4.5.2 Configure a repository through the GraphDB Workbench ... 75
  4.5.3 Edit a repository ............................................... 75
  4.5.4 Configure a repository programmatically .................... 76
  4.5.5 Configuration parameters ...................................... 78
  4.5.6 Configure GraphDB memory ..................................... 80
    4.5.6.1 Configure Java heap memory ............................... 80
    4.5.6.2 Single global page cache ................................ 80
    4.5.6.3 Configure Entity pool memory ............................ 80
    4.5.6.4 Sample memory configuration .............................. 81
    4.5.6.5 Upper bounds for the memory consumed by the GraphDB process ........................................... 81
  4.5.7 Reconfigure a repository ........................................ 81
    4.5.7.1 Using the Workbench ...................................... 81
    4.5.7.2 Global overrides .......................................... 82

4.5.8 Rename a repository ............................................... 82
    4.5.8.1 Using the Workbench ...................................... 82

4.6 Creating and Managing a Cluster ................................... 82
  4.6.1 Prerequisites .................................................... 82
  4.6.2 High availability deployment .................................. 82
    4.6.2.1 Cluster group with three nodes .......................... 83
    4.6.2.2 Cluster group with five nodes ............................ 83
  4.6.3 Create cluster .................................................. 83
    4.6.3.1 Using the Workbench ...................................... 83
    4.6.3.2 Using the REST API ........................................ 86
    4.6.3.3 Creation parameters ....................................... 87
  4.6.4 Manage cluster membership ..................................... 87
    4.6.4.1 Add nodes .................................................. 88
    4.6.4.2 Remove nodes .............................................. 88
  4.6.5 Manage cluster configuration properties ...................... 90
    4.6.5.1 Using the Workbench ...................................... 90
    4.6.5.2 Using the REST API ........................................ 91
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.6 Monitor cluster status</td>
<td>91</td>
</tr>
<tr>
<td>4.6.6.1 Cluster group</td>
<td>92</td>
</tr>
<tr>
<td>4.6.6.2 Cluster node</td>
<td>93</td>
</tr>
<tr>
<td>4.6.7 Delete cluster</td>
<td>93</td>
</tr>
<tr>
<td>4.6.7.1 Force parameter</td>
<td>94</td>
</tr>
<tr>
<td>4.6.8 Configure external cluster proxy</td>
<td>94</td>
</tr>
<tr>
<td>4.6.8.1 Start the external proxy</td>
<td>95</td>
</tr>
<tr>
<td>4.6.8.2 External proxy for cluster running over SSL</td>
<td>96</td>
</tr>
<tr>
<td>4.6.9 Cluster security</td>
<td>96</td>
</tr>
<tr>
<td>4.6.9.1 Encryption</td>
<td>96</td>
</tr>
<tr>
<td>4.6.9.2 Access control</td>
<td>96</td>
</tr>
<tr>
<td>4.7 Using a Cluster</td>
<td>96</td>
</tr>
<tr>
<td>4.7.1 Using the GraphDB client API for Java</td>
<td>97</td>
</tr>
<tr>
<td>4.7.2 Using a cluster with external proxy</td>
<td>98</td>
</tr>
<tr>
<td>4.8 Secure GraphDB</td>
<td>99</td>
</tr>
<tr>
<td>4.8.1 Enable security</td>
<td>99</td>
</tr>
<tr>
<td>4.8.2 Login and default credentials</td>
<td>100</td>
</tr>
<tr>
<td>4.8.3 Free access</td>
<td>100</td>
</tr>
<tr>
<td>4.8.4 Users and Roles</td>
<td>100</td>
</tr>
<tr>
<td>4.8.4.1 Create new user</td>
<td>100</td>
</tr>
<tr>
<td>4.8.4.2 Set password</td>
<td>101</td>
</tr>
<tr>
<td>4.9 Request Tracking</td>
<td>102</td>
</tr>
<tr>
<td>4.10 Application Settings</td>
<td>102</td>
</tr>
<tr>
<td>4.11 Backing up and Restoring a Repository</td>
<td>103</td>
</tr>
<tr>
<td>4.11.1 Back up a repository</td>
<td>103</td>
</tr>
<tr>
<td>4.11.1.1 Export repository to an RDF file</td>
<td>103</td>
</tr>
<tr>
<td>4.11.1.2 Back up a repository using JMX interface</td>
<td>105</td>
</tr>
<tr>
<td>4.11.1.3 Back up GraphDB by copying the binary image</td>
<td>107</td>
</tr>
<tr>
<td>4.11.2 Restore a repository</td>
<td>107</td>
</tr>
<tr>
<td>4.12 Query Monitoring and Termination</td>
<td>108</td>
</tr>
<tr>
<td>4.12.1 Query monitoring and termination using the Workbench</td>
<td>108</td>
</tr>
<tr>
<td>4.12.2 Query monitoring and termination using the JMX interface</td>
<td>109</td>
</tr>
<tr>
<td>4.12.2.1 Query monitoring</td>
<td>109</td>
</tr>
<tr>
<td>4.12.2.2 Terminating a query</td>
<td>110</td>
</tr>
<tr>
<td>4.12.3 Terminating a transaction</td>
<td>111</td>
</tr>
<tr>
<td>4.12.4 Automatically prevent long running queries</td>
<td>111</td>
</tr>
<tr>
<td>4.13 Performance Optimizations</td>
<td>111</td>
</tr>
<tr>
<td>4.13.1 Data loading &amp; query optimizations</td>
<td>112</td>
</tr>
<tr>
<td>4.13.1.1 Dataset loading</td>
<td>112</td>
</tr>
<tr>
<td>4.13.1.2 GraphDB’s optional indexes</td>
<td>113</td>
</tr>
<tr>
<td>4.13.1.3 Cache/index monitoring and optimizations</td>
<td>114</td>
</tr>
<tr>
<td>4.13.1.4 Query optimizations</td>
<td>115</td>
</tr>
<tr>
<td>4.13.1.5 Index compacting</td>
<td>116</td>
</tr>
<tr>
<td>4.13.2 Explain Plan</td>
<td>116</td>
</tr>
<tr>
<td>4.13.2.1 What is GraphDB’s Explain Plan</td>
<td>116</td>
</tr>
<tr>
<td>4.13.2.2 Activating the explain plan</td>
<td>116</td>
</tr>
<tr>
<td>4.13.2.3 Simple explain plan</td>
<td>117</td>
</tr>
<tr>
<td>4.13.2.4 Multiple triple patterns</td>
<td>118</td>
</tr>
<tr>
<td>4.13.2.5 Wine queries</td>
<td>119</td>
</tr>
<tr>
<td>4.13.3 Inference optimizations</td>
<td>121</td>
</tr>
<tr>
<td>4.13.3.1 Delete optimizations</td>
<td>121</td>
</tr>
<tr>
<td>4.13.3.2 Rules optimizations</td>
<td>123</td>
</tr>
<tr>
<td>4.13.3.3 Optimization of owl:sameAs</td>
<td>133</td>
</tr>
<tr>
<td>4.13.3.4 RDFS and OWL support optimizations</td>
<td>139</td>
</tr>
<tr>
<td>4.14 Troubleshooting</td>
<td>139</td>
</tr>
<tr>
<td>4.14.1 Database health checks</td>
<td>139</td>
</tr>
<tr>
<td>4.14.1.1 Possible values for health checks and their meaning</td>
<td>140</td>
</tr>
<tr>
<td>4.14.1.2 Aggregated health checks</td>
<td>140</td>
</tr>
</tbody>
</table>
5 Usage

5.1 Loading Data ......................................................... 151
  5.1.1 Loading data using the Workbench ........................................... 151
    5.1.1.1 Import settings ................................................. 152
    5.1.1.2 Importing local files ........................................... 153
    5.1.1.3 Importing remote content ....................................... 154
    5.1.1.4 Importing RDF data from a text snippet .......................... 154
    5.1.1.5 Importing server files ......................................... 154
    5.1.1.6 Import data with an INSERT query ................................ 155
  5.1.2 Loading data using the ImportRDF tool .................................. 155
    5.1.2.1 Load vs Preload ................................................... 155
    5.1.2.2 Load command line options ..................................... 156
    5.1.2.3 Preload command line options ................................... 156
    5.1.2.4 Repository configuration template ................................ 157
    5.1.2.5 Tuning Load ...................................................... 158
    5.1.2.6 Tuning Preload .................................................. 158
    5.1.2.7 Resuming data loading with Preload ............................... 159

5.2 Exploring data ......................................................... 159
  5.2.1 Class hierarchy ................................................... 159
    5.2.1.1 Explore your data - different actions ........................... 160
    5.2.1.2 Domain-range graph ............................................. 163
  5.2.2 Class relationships ................................................ 163
  5.2.3 Explore resources .................................................. 168
    5.2.3.1 Explore resources through the easy graph ....................... 168
    5.2.3.2 Create your own visual graph ................................... 176
    5.2.3.3 Save and share graphs .......................................... 177
  5.2.4 View and edit resources ............................................. 178
    5.2.4.1 View and add a resource ....................................... 178
    5.2.4.2 Edit a resource .................................................. 180

5.3 Querying Data ........................................................ 180
  5.3.1 Save and share queries ............................................. 182
  5.3.2 Interrupt queries ................................................... 183

5.4 Exporting Data ......................................................... 183
  5.4.1 Exporting a repository .............................................. 183
  5.4.2 Exporting individual graphs ......................................... 183
  5.4.3 Exporting query results ............................................. 184
  5.4.4 Exporting resources ................................................ 184

5.5 Updating Data ........................................................ 185
  5.5.1 Overview ........................................................... 185
    5.5.1.1 Replace graph ................................................... 185
    5.5.1.2 DELETE/INSERT template ....................................... 185
  5.5.2 Transport mechanisms ............................................... 186
    5.5.2.1 Direct access ................................................... 186
    5.5.2.2 Indirect access via Kafka Sink connector ....................... 187
  5.5.3 SPARQL templates .................................................. 188
    5.5.3.1 From the SPARQL editor ......................................... 188
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.3.2 From the SPARQL Templates editor</td>
<td>190</td>
</tr>
<tr>
<td>5.5.3.3 SPARQL Template endpoint</td>
<td>190</td>
</tr>
<tr>
<td>5.6 Kafka Sink Connector</td>
<td>194</td>
</tr>
<tr>
<td>5.6.1 Overview</td>
<td>194</td>
</tr>
<tr>
<td>5.6.2 Setup</td>
<td>194</td>
</tr>
<tr>
<td>5.6.3 Update types</td>
<td>194</td>
</tr>
<tr>
<td>5.6.3.1 Simple add</td>
<td>195</td>
</tr>
<tr>
<td>5.6.3.2 Replace graph</td>
<td>196</td>
</tr>
<tr>
<td>5.6.3.3 DELETE/INSERT template</td>
<td>198</td>
</tr>
<tr>
<td>5.6.4 Configuration properties</td>
<td>200</td>
</tr>
<tr>
<td>5.7 Reasoning</td>
<td>202</td>
</tr>
<tr>
<td>5.7.1 Logical formalism</td>
<td>202</td>
</tr>
<tr>
<td>5.7.2 Rule format and semantics</td>
<td>202</td>
</tr>
<tr>
<td>5.7.3 The ruleset file</td>
<td>203</td>
</tr>
<tr>
<td>5.7.3.1 Prefixes</td>
<td>203</td>
</tr>
<tr>
<td>5.7.3.2 Axioms</td>
<td>203</td>
</tr>
<tr>
<td>5.7.3.3 Rules</td>
<td>204</td>
</tr>
<tr>
<td>5.7.4 Rulesets</td>
<td>207</td>
</tr>
<tr>
<td>5.7.4.1 Predefined rulesets</td>
<td>208</td>
</tr>
<tr>
<td>5.7.4.2 Custom rulesets</td>
<td>209</td>
</tr>
<tr>
<td>5.7.5 Inference</td>
<td>209</td>
</tr>
<tr>
<td>5.7.5.1 Reasoner</td>
<td>209</td>
</tr>
<tr>
<td>5.7.5.2 Rulesets execution</td>
<td>209</td>
</tr>
<tr>
<td>5.7.5.3 Retraction of assertions</td>
<td>210</td>
</tr>
<tr>
<td>5.7.6 How To's</td>
<td>211</td>
</tr>
<tr>
<td>5.7.6.1 Operations on rulesets</td>
<td>211</td>
</tr>
<tr>
<td>5.7.6.2 Reinferring</td>
<td>214</td>
</tr>
<tr>
<td>5.7.7 Provenance</td>
<td>215</td>
</tr>
<tr>
<td>5.8 SHACL Validation</td>
<td>215</td>
</tr>
<tr>
<td>5.8.1 What is SHACL validation?</td>
<td>215</td>
</tr>
<tr>
<td>5.8.2 Usage</td>
<td>215</td>
</tr>
<tr>
<td>5.8.2.1 Creating and configuring a SHACL repository</td>
<td>215</td>
</tr>
<tr>
<td>5.8.2.2 Loading shapes and data graphs</td>
<td>216</td>
</tr>
<tr>
<td>5.8.2.3 Deleting shapes and data graphs</td>
<td>217</td>
</tr>
<tr>
<td>5.8.2.4 Updating shapes and data graphs</td>
<td>218</td>
</tr>
<tr>
<td>5.8.2.5 Viewing shapes and data graphs</td>
<td>218</td>
</tr>
<tr>
<td>5.8.3 Validation logging and report</td>
<td>218</td>
</tr>
<tr>
<td>5.8.4 Supported SHACL features</td>
<td>219</td>
</tr>
<tr>
<td>5.9 Virtualization</td>
<td>220</td>
</tr>
<tr>
<td>5.9.1 Overview and features</td>
<td>220</td>
</tr>
<tr>
<td>5.9.2 Usage scenario</td>
<td>221</td>
</tr>
<tr>
<td>5.9.3 Setup and configuration</td>
<td>223</td>
</tr>
<tr>
<td>5.9.3.1 JDBC driver</td>
<td>223</td>
</tr>
<tr>
<td>5.9.3.2 Configuration files</td>
<td>223</td>
</tr>
<tr>
<td>5.9.3.3 Creating a virtual repository from the Workbench</td>
<td>223</td>
</tr>
<tr>
<td>5.9.3.4 Creating a virtual repository using cURL</td>
<td>225</td>
</tr>
<tr>
<td>5.9.4 Mapping language</td>
<td>225</td>
</tr>
<tr>
<td>5.9.5 SPARQL endpoint</td>
<td>226</td>
</tr>
<tr>
<td>5.9.6 Query federation</td>
<td>227</td>
</tr>
<tr>
<td>5.9.7 Limitations</td>
<td>228</td>
</tr>
<tr>
<td>5.10 Using the GraphDB REST API</td>
<td>228</td>
</tr>
<tr>
<td>5.10.1 Cluster group controller</td>
<td>228</td>
</tr>
<tr>
<td>5.10.2 Data import</td>
<td>229</td>
</tr>
<tr>
<td>5.10.3 Location management</td>
<td>229</td>
</tr>
<tr>
<td>5.10.4 Recovery management</td>
<td>229</td>
</tr>
<tr>
<td>5.10.5 Repository management</td>
<td>229</td>
</tr>
<tr>
<td>5.10.6 Saved queries</td>
<td>229</td>
</tr>
<tr>
<td>5.10.7 Security management</td>
<td>229</td>
</tr>
</tbody>
</table>
5.10.8 SPARQL templates ........................................... 229
5.10.9 SQL views management ..................................... 230
5.10.10 Authentication .............................................. 230

5.11 Using GraphDB with the RDF4J API ............................. 231
5.11.1 RDF4J API ....................................................... 231
   5.11.1.1 Accessing a local repository .......................... 231
   5.11.1.2 Accessing a remote repository ....................... 232
5.11.2 SPARQL endpoint ............................................ 233
5.11.3 Graph Store HTTP Protocol ................................. 233

5.12 SQL Access over JDBC .......................................... 233
5.12.1 Configuration ............................................... 233
   5.12.1.1 Prerequisites .......................................... 233
   5.12.1.2 Creating a SQL view .................................. 234
   5.12.1.3 Updating a SQL view ................................ 237
   5.12.1.4 Deleting a SQL view .................................. 237
5.12.2 Type mapping ................................................ 237
5.12.3 WHERE to FILTER conversion ................................. 238
5.12.4 Table verification ............................................ 238
5.12.5 Usage examples .............................................. 240
   5.12.5.1 Tableau .................................................. 240
   5.12.5.2 Microsoft Power BI over ODBC protocol ............ 242
5.12.6 How it works: Table over ODBC protocol .................... 244

5.13 Plugins ........................................................ 245
5.13.1 Activate & enable plugins ................................... 245
   5.13.1.1 Activate/deactivate plugins .......................... 245
   5.13.1.2 Enable/disable plugins .............................. 246
5.13.2 Plugin API ..................................................... 247
   5.13.2.1 What is the GraphDB Plugin API ...................... 247
   5.13.2.2 Description of a GraphDB plugin .................... 247
   5.13.2.3 The life cycle of a plugin ............................ 249
   5.13.2.4 Repository internals ................................. 250
   5.13.2.5 Query processing ..................................... 255
   5.13.2.6 Update processing .................................... 260
   5.13.2.7 Transactions .......................................... 263
   5.13.2.8 Exceptions ............................................. 264
   5.13.2.9 Accessing other plugins ............................. 265
   5.13.2.10 List of plugin interfaces and classes ............... 266
   5.13.2.11 Adding external plugins to GraphDB ............... 270
   5.13.2.12 Putting it all together: example plugins .......... 271

5.13.3 RDF Rank ..................................................... 276
   5.13.3.1 What is RDF Rank ..................................... 276
   5.13.3.2 Parameters ............................................ 277
   5.13.3.3 Full computation ..................................... 277
   5.13.3.4 Incremental updates .................................. 278
   5.13.3.5 Exporting RDF Rank values ............................ 278
   5.13.3.6 Checking the RDF Rank status ........................ 278
   5.13.3.7 Rank filtering ........................................ 279

5.13.4 Semantic similarity searches ............................... 280
   5.13.4.1 Why do I need the similarity plugin? ................ 280
   5.13.4.2 What the similarity plugin does? ........................ 280
   5.13.4.3 How the similarity plugin works? ...................... 280
   5.13.4.4 Download data ........................................ 284
   5.13.4.5 Text based similarity searches ....................... 286
   5.13.4.6 Predication-based Semantic Indexing ................. 294
   5.13.4.7 Hybrid indexing ..................................... 298
   5.13.4.8 Training cycles ....................................... 300

5.13.5 JavaScript functions ......................................... 301
   5.13.5.1 How to register a JS function ........................ 301
7 Developer Hub

7.1 Data Modeling with RDF(S) .................................................. 595
  7.1.1 What is RDF? .............................................................. 595
  7.1.2 What is RDFS? ............................................................ 596

7.2 SPARQL ................................................................. 597
  7.2.1 What is SPARQL? ....................................................... 597
  7.2.2 Using SPARQL in GraphDB ......................................... 599

7.3 RDF-star and SPARQL-star ............................................... 599
  7.3.1 The modeling challenge ............................................. 599
    7.3.1.1 Standard reification ........................................ 600
    7.3.1.2 N-ary relations ............................................. 600
    7.3.1.3 Singleton properties ....................................... 600

6 Security

6.1 Access Control .......................................................... 573
  6.1.1 Authorization and user database ..................................... 573
    6.1.1.1 User roles and permissions .................................. 573
    6.1.1.2 Built-in users and roles .................................... 575
    6.1.1.3 Local user database ......................................... 575
    6.1.1.4 LDAP user database ......................................... 576
    6.1.1.5 OAuth user database ........................................ 578
  6.1.2 Authentication methods ............................................. 579
    6.1.2.1 Basic authentication ....................................... 581
    6.1.2.2 GDB authentication ......................................... 581
    6.1.2.3 OpenID authentication ..................................... 582
    6.1.2.4 Kerberos authentication .................................... 584
  6.1.3 Example configurations ............................................. 586
    6.1.3.1 Basic/GDB + LDAP ......................................... 586
    6.1.3.2 OpenID + Local users ................................ ..... 587
    6.1.3.3 OpenID + LDAP ............................................... 587
    6.1.3.4 OpenID + OAuth ............................................. 588
    6.1.3.5 Kerberos + Local users ..................................... 589
    6.1.3.6 Kerberos + LDAP ............................................. 589
  6.2 Encryption ............................................................. 590
    6.2.1 Encryption in transit ............................................ 590
      6.2.1.1 Configuring GraphDB instance with SSL ................... 590
      6.2.1.2 Certificate trust ......................................... 591
    6.2.2 Encryption at rest ................................................ 592
  6.3 Security Auditing .................................................... 592
7.10 Create Custom Graph View over Your RDF Data
7.10.1 How it works? .......................... 660
7.10.2 World airport, airline, and route data..... 660
    7.10.2.1 Data model ....................... 660
7.9 Visualize GraphDB Data with Ognaj
7.9.1 People and organizations related to Google in factforge.net ................................................................................... 647
7.9.2 Suspicious control chain through off-shore companies in factforge.net ...................................................... 651
7.9.3 Shortest flight path ................................ 654
7.9.4 Common function to visualize GraphDB data ......................................................... 658
7.8 GraphDB REST API ................................ 630
7.8.1 Location and repository management with the GraphDB REST API .......................... 630
    7.8.1.1 Prerequisites .......................... 630
    7.8.1.2 Managing repositories ............... 630
    7.8.1.3 Managing locations ................ 631
    7.8.1.4 Further reading ........................ 632
7.8.2 Cluster management with the GraphDB REST API ........................................... 632
7.8.3 GraphDB REST API cURL commands ......................................................... 633
    7.8.3.1 Cluster group management ......... 633
    7.8.3.2 Data import .......................... 634
    7.8.3.3 Location management ................ 636
    7.8.3.4 Recovery management ............. 637
    7.8.3.5 Repository management .......... 638
    7.8.3.6 Saved queries ....................... 642
    7.8.3.7 Security management ............... 643
    7.8.3.8 SPARQL template management .... 645
    7.8.3.9 SQL views management ............ 646
    7.8.3.10 Authentication .................... 647
7.7 Extending GraphDB Workbench ................ 619
    7.7.1 Clone, download, and run GraphDB Workbench ........................................ 619
    7.7.2 Add your own page and controller ................................................................. 619
    7.7.3 Add repository checks .................................................................................. 620
    7.7.4 Repository setup .......................... 621
    7.7.5 Select departure and destination airport ...................................................... 621
    7.7.6 Find the paths between the selected airports .............................................. 623
    7.7.7 Visualize results ......................................................................................... 629
    7.7.8 Add status message .................................................................................... 629
7.6 Programming with GraphDB .................... 610
    7.6.1 Installing Maven dependencies ................................................................. 611
    7.6.2 Examples .................................................................................................... 611
        7.6.2.1 Hello world in GraphDB ......... 611
        7.6.2.2 Family relations app .............. 612
        7.6.2.3 Embedded GraphDB ............. 616
7.5 Inference ............................................ 609
    7.5.1 What is inference? ....................................................................................... 609
    7.5.2 Inference in GraphDB .................................................................................. 609
        7.5.2.1 Standard rule sets ................. 610
        7.5.2.2 Custom rule sets ................... 610
7.4 Ontologies ........................................... 607
    7.4.1 What is an ontology? .................................................................................... 607
    7.4.2 What are the benefits of developing and using an ontology? ....................... 608
    7.4.3 Using ontologies in GraphDB ....................................................................... 609
7.3 MIME types and file extensions for RDF-star in RDF4J ........................................ 601
    7.3.1 RDF-star and SPARQL-star ......................................................................... 601
    7.3.2 How the different approaches compare? ..................................................... 602
    7.3.3 Syntax and examples .................................................................................... 602
    7.3.3.1 Embedded triple visualization ................................................... 604
    7.3.4 Convert standard reification to RDF-star .................................................... 606
    7.3.5 MIME types and file extensions for RDF-star in RDF4J ............................... 607
7.2 Inference in the real world: developing a rule set .............................................. 601
7.1 Visualize GraphDB Data with Ognaj .................................................................. 601
8.4 Ontology Mapping with owl:sameAs Property .................................................. 692
8.5 SPARQL Compliance ......................................................................................... 694
  8.5.1 SPARQL 1.1 Protocol for RDF ................................................................. 694
  8.5.2 SPARQL 1.1 Query .................................................................................... 694
  8.5.3 SPARQL 1.1 Update .................................................................................. 694
    8.5.3.1 Modification operations on the RDF triples ....................................... 694
    8.5.3.2 Operations for managing graphs ....................................................... 695
  8.5.4 SPARQL 1.1 Federation ........................................................................... 695
    8.5.4.1 Internal SPARQL federation ............................................................. 695
    8.5.4.2 Federated query to a remote password-protected repository ............ 695
  8.5.5 SPARQL 1.1 Graph Store HTTP Protocol ................................................. 695
    8.5.5.1 URL patterns for this new functionality are provided at .................. 695
    8.5.5.2 Methods supported by these resources and their effects ................. 696
    8.5.5.3 Request headers ................................................................................ 696
    8.5.5.4 Supported parameters for requests on indirectly referenced named graphs 696
  8.6 SPARQL Functions Reference ....................................................................... 696
    8.6.1 SPARQL 1.1 functions ........................................................................... 697
    8.6.2 SPARQL 1.1 constructor functions ........................................................ 701
    8.6.3 Mathematical function extensions .......................................................... 702
    8.6.4 Date and time function extensions ......................................................... 705
    8.6.5 SPARQL SPIN functions and magic predicates ....................................... 707
      8.6.5.1 SPARQL functions ........................................................................ 709
      8.6.5.2 Magic predicates ........................................................................... 710
    8.6.6 RDF-star extension functions ................................................................. 710
    8.6.7 GeoSPARQL functions .......................................................................... 710
      8.6.7.1 GeoSPARQL extension functions .................................................. 713
    8.6.8 Geospatial extension query functions ..................................................... 713
  8.7 OWL Compliance ............................................................................................ 714
  8.8 Glossary .......................................................................................................... 714

9 Release Notes ...................................................................................................... 717
  9.1 GraphDB 10.0.2 ............................................................................................. 717
    9.1.1 Component versions ............................................................................... 717
    9.1.2 GraphDB Connectors & Plugins ............................................................ 718
      9.1.2.1 Bug fixing ....................................................................................... 718
  9.2 GraphDB 10.0.1 ............................................................................................. 718
    9.2.1 Component versions ............................................................................... 718
    9.2.2 GraphDB Engine & Cluster .................................................................. 718
      9.2.2.1 New features and improvements .................................................. 718
      9.2.2.2 Bug fixing ....................................................................................... 718
    9.2.3 GraphDB Workbench ............................................................................ 719
      9.2.3.1 Bug fixing ....................................................................................... 719
    9.2.4 GraphDB Connectors & Plugins ............................................................ 719
      9.2.4.1 New features and improvements .................................................. 719
  9.2.5 GraphDB Distributions & Deployment ..................................................... 719
    9.2.5.1 New features and improvements .................................................. 719
    9.2.5.2 Bug fixing ....................................................................................... 719
  9.3 GraphDB 10.0.0 ............................................................................................. 720
    9.3.1 Component versions ............................................................................... 720
    9.3.2 GraphDB Engine & Cluster .................................................................. 721
      9.3.2.1 New features and improvements .................................................. 721
      9.3.2.2 Bug fixing ....................................................................................... 722
    9.3.3 GraphDB Workbench ............................................................................ 722
      9.3.3.1 New features and improvements .................................................. 722
      9.3.3.2 Bug fixing ....................................................................................... 723
    9.3.4 GraphDB Connectors & Plugins ............................................................ 723
      9.3.4.1 New features and improvements .................................................. 723
      9.3.4.2 Bug fixing ....................................................................................... 723
**Ontotext GraphDB** is a highly efficient and robust graph database with RDF and SPARQL support. This documentation is a comprehensive guide that explains every feature of GraphDB, as well as topics such as setting up a repository, loading and working with data, tuning its performance, scaling, etc.

The GraphDB database supports a highly available replication cluster, which has been proven in a number of enterprise use cases that required resilience in data loading and query answering. If you need a quick overview of GraphDB or a download link to its latest releases, please visit the GraphDB product section.

**Credits and licensing**

GraphDB uses RDF4J as a library, utilizing its APIs for storage and querying, as well as the support for a wide variety of query languages (e.g., SPARQL and SeRQL) and RDF syntaxes (e.g., RDF/XML, N3, Turtle).

Full licensing information is available in the license files located in the doc folder of the distribution package.

**Helpful hints**

Throughout the documentation there are a number of helpful pieces of information that can give you additional information, warn you, or save you time and unnecessary effort. Here is what to pay attention to:

**Hint:** Hint badges give additional information you may find useful.

**Tip:** Tip badges are handy pieces of information.

**Note:** Notes are comments or references that may save you time and unnecessary effort.
1.1 About GraphDB

What makes GraphDB different?

- Manages tens of billions of RDF statements on a single server
- Performs query and reasoning operations using file-based indexes
- Full SPARQL 1.1 support
- Easy Java deployment and portability
- Scalability, both in terms of the data volume and loading and inferencing speed
- High performance load, query, and inference simultaneously
- Compatible with RDF4J 2.0
- Compatible with Jena with a built-in adapter
- Full standard-compliant reasoning for RDFS, OWL 2 RL, and QL
- Support for custom reasoning rulesets, performance optimized rulesets
- Optimized support for data integration through owl:sameAs
- Special indexes for efficient geospatial constraints (near-by, within, distance)
- Efficient retraction of inferred statements upon update
- Reliable data preservation, consistency, and integrity
- Import/export of RDF syntaxes through RDF4J: XML, N3, N-Triples, N-Quads, Turtle, TriG, TriX
- API plugin framework, public classes and interfaces
- Query optimizer allowing for the evaluation of different query plans
- RDF rank to order query results by relevance or other measures
- Notifications allowing clients to react to statements in the update stream
- Lucene connector, Solr, and Elasticsearch for extremely fast normal and faceted (aggregation) searches; automatically stay up-to-date with the GraphDB data
- GraphDB Workbench - the default web-based administration tool
- ImportRDF for very fast repository creation from big datasets
- Transparent upgrade from GraphDB SE or Free edition
- Kafka connector for synchronizing changes to the RDF model to any Kafka consumer
- High-availability cluster based on the Raft consensus algorithm, designed for high availability, with several features that are crucial for achieving enterprise-grade highly available deployments.

GraphDB is a family of highly efficient, robust, and scalable RDF databases. It streamlines the load and use of linked data cloud datasets, as well as your own resources. For easy use and compatibility with the industry standards, GraphDB implements the RDF4J framework interfaces, the W3C SPARQL Protocol specification, and supports all RDF serialization formats. The database is the preferred choice of both small independent developers and big enterprise organizations because of its community and commercial support, as well as excellent enterprise features such as cluster support and integration with external high-performance search applications - Lucene, Solr, and Elasticsearch.
GraphDB is one of the few triplestores that can perform semantic inferencing at scale, allowing users to derive new semantic facts from existing facts. It handles massive loads, queries, and inferencing in real time. 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 a desktop hardware and handle tens of billions of statements on a 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 three editions of GraphDB:

- GraphDB Free
- GraphDB Standard (SE)
- GraphDB Enterprise (EE)

1.1.1 Comparison of GraphDB Free, GraphDB Standard, and GraphDB Enterprise

GraphDB Free and GraphDB SE are identical in terms of usage and integration and share most features: they are both designed as an enterprise-grade semantic repository system, are suitable for massive volumes of data, employ file-based indexes that enable them to scale to billions of statements even on desktop machines, and ensure fast query evaluations through inference and query optimizations.

GraphDB Free is commercial & free to use, supports a limit of two concurrent queries, and is suitable for low query loads and smaller projects.

GraphDB SE is commercial, supports an unlimited number of concurrent queries, and is suitable for heavy query loads.

Building up on the above, the GraphDB EE edition is a high-performance, clustered semantic repository scaling 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, designed for high availability, 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.

To find out more about the differences between the editions, see the GraphDB Feature Comparison section.

1.2 Architecture & Components

1.2.1 Architecture

GraphDB is packaged as a SAIL (Storage And Inference Layer) for RDF4J and makes extensive use of the features and infrastructure of RDF4J, especially the RDF model, RDF parsers, and query engines.

Inference is performed by the 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.
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 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.

1.2.1.2 The Sail API

The Sail API is a set of Java interfaces that support RDF storing, retrieving, deleting, and inferencing. It is used for abstracting from the actual storage mechanism, e.g., an implementation can use relational databases, file systems, in-memory storage, etc. One of its key characteristics is the option for SAIL stacking.

1.2.2 Components

1.2.2.1 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 where optimization occurs. GraphDB allows hints for guiding the query optimizer.

Reasoner (TRREE Engine)

GraphDB is implemented on top of the TRREE engine. TRREE stands for ‘Triple Reasoning and Rule Entailment Engine’. The 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 PS0, context index CPS0, and literal index, with the latter two being optional.

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 worker 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.

1.2.2.2 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)

Additionally, the Kafka GraphDB connector provides a means to synchronize changes to the RDF model to any Kafka consumer. (requires a GraphDB Enterprise license)

1.2.2.3 Workbench

The Workbench is the GraphDB web-based administration tool.

1.3 Cluster Basics

1.3.1 High availability features

A system can be characterized as having high availability if it meets several key criteria such as: having a high uptime, recovering smoothly, achieving zero data loss, and essentially handling and adapting to unexpected situations and scenarios.

The GraphDB cluster is designed for high availability and has several features that are crucial for achieving enterprise-grade highly available deployments. It is based on coordination mechanisms known as consensus algorithms. They allow a collection of machines to work as a coherent group that can survive the failures of some of its members and provide lower latency. In essence, such protocols define the set of rules for messaging between machines. Because of this, they play a key role in building reliable large-scale software systems.

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.

The GraphDB cluster uses the Raft consensus algorithm for managing a replicated log on distributed state machines. It implements consensus by first electing a distinguished leader, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. It can tolerate \( n \geq 2m + 1 \) failures.
1.3.1.1 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/DELETE operations. This ensures that there will always be a majority of GraphDB nodes that always have up-to-date data.

If there are unavailable nodes when an INSERT/DELETE operation is executed, but there are more than 50% alive nodes, 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 there are fewer than 50% available nodes, any INSERT/DELETE operations will be rejected.

1.3.1.2 Internal and external proxy

**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, you need to switch to another cluster node that will be on another URL. This means that you need to know all cluster node addresses and make sure that the reached node is healthy and online.

**External proxy**

For even better usability, the 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 one the leader is and will always serve all requests to the current leader.

1.3.1.3 Query load balancer

In order to achieve maximum efficiency, the GraphDB cluster distributes the incoming read queries to all nodes, prioritizing the ones that have fewer running queries. This ensures the optimal hardware resource utilization of all nodes.

1.3.2 Cluster roles

As mentioned above, the GraphDB 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, i.e., 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, i.e., invalid or expired license.
1.3.3 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 will result 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 passing. 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.

1.3.4 Log replication

The GraphDB cluster nodes communicate using remote procedure calls (RPCs), and 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, and 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 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 start applying the entry to GraphDB successfully.
6. The leader sends a heartbeat until successful commit in GraphDB.
7. The leader sends a second RPC informing followers to apply 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 adding/deleting repositories, user right changes, SQL views, smart updates, and standard repository updates.

1.3.5 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 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 called 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 goes as follows:

1. After the initial configuration request has been sent, one of the nodes will be set as leader at random.
2. If the current leader node stops working for some reason, a new election is being 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 and it will switch to read-only mode to prevent further tinkering with it, until more nodes appear in the cluster group.
1.4 Connectors

The GraphDB Connectors enable the connection to an external component or service, providing full-text search and aggregation (Lucene, Solr, Elasticsearch), or querying a database using SPARQL and executing heterogeneous joins (MongoDB). They also offer the additional benefit of staying automatically up-to-date with the GraphDB repository data.

1.4.1 Full-text search and aggregation connectors

The Lucene, Solr, and Elasticsearch Connectors provide synchronization at entity level, where an entity is defined as having a unique identifier (URI) and a set of properties and property values. In RDF context, 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 subsequent triple.

GraphDB comes with the following FTS connector implementations:

- Lucene GraphDB connector
- Solr GraphDB connector (requires a GraphDB Enterprise license)
- Elasticsearch GraphDB connector (requires a GraphDB Enterprise license)
Table 1: Features Comparison Table: FTS and Aggregation Connectors

<table>
<thead>
<tr>
<th>Feature</th>
<th>Lucene</th>
<th>Solr</th>
<th>Elasticsearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTS search</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simple facets</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sorting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Snippet extraction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Limit and offset</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fixed range facets</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Variable range facets</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nested facets</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregations</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>• terms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• histogram</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• range</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• min/max</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• sum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• average</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• standard deviation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• variance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• sum of squares</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sub-aggregations        | ×      | ×    | ✓             |

1.4.2 MongoDB integration

The MongoDB integration allows you to query MongoDB databases using SPARQL and to execute heterogeneous joins. A document-based database with the biggest developer/user community, 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 RDF model.

1.4.3 Kafka connector

The Kafka GraphDB connector provides a means to synchronize changes to 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 a reliable synchronization mechanism is needed.

This functionality requires a GraphDB Enterprise license.

**Note:** Despite having a similar name, the Kafka Sink connector is not a GraphDB connector.

1.5 Workbench

Workbench is the GraphDB web-based administration tool. The user interface is similar to the RDF4J Workbench Web Application, but with more functionality.

**What makes GraphDB Workbench different?**

- Better SPARQL editor based on YASGUI
The GraphDB Workbench can be used for:

- managing GraphDB repositories;
- loading and exporting data;
- executing SPARQL queries and updates;
- managing namespaces;
- managing contexts;
- viewing/editing RDF resources;
- monitoring queries;
- monitoring resources;
- managing users and permissions;
- managing connectors;
- managing a cluster;
- viewing license information;
- providing REST API for automating various tasks for managing and administrating repositories.

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 Extend GraphDB Workbench.

## 1.6 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.

### 1.6.1 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.
1.6.1.1 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

<table>
<thead>
<tr>
<th>Editions</th>
<th>Ruleset</th>
<th>Explicit statements</th>
<th>Total statements</th>
<th>AWS instance</th>
<th>Cores</th>
<th>Loading time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0 Free</td>
<td>RDFS-Plus-optimized</td>
<td>237,802,643</td>
<td>385,168,491</td>
<td>i3.xlarge</td>
<td>1*</td>
<td>399</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>RDFS-Plus-optimized</td>
<td>237,802,643</td>
<td>385,168,491</td>
<td>i3.xlarge</td>
<td>2</td>
<td>315</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>RDFS-Plus-optimized</td>
<td>237,802,643</td>
<td>385,168,491</td>
<td>i3.xlarge</td>
<td>4</td>
<td>312</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>RDFS-Plus-optimized</td>
<td>237,802,643</td>
<td>385,168,491</td>
<td>i3.2xlarge</td>
<td>8</td>
<td>259</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>RDFS-Plus-optimized</td>
<td>237,802,643</td>
<td>385,168,491</td>
<td>i3.4xlarge</td>
<td>16</td>
<td>253</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Editions</th>
<th>Ruleset</th>
<th>Explicit statements</th>
<th>Total statements</th>
<th>AWS instance</th>
<th>Cores</th>
<th>Loading time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0 SE/EE</td>
<td>OWL2-RL</td>
<td>237,802,643</td>
<td>752,341,659</td>
<td>i3.xlarge</td>
<td>2</td>
<td>889</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>OWL2-RL</td>
<td>237,802,643</td>
<td>752,341,659</td>
<td>i3.xlarge</td>
<td>4</td>
<td>843</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>OWL2-RL</td>
<td>237,802,643</td>
<td>752,341,659</td>
<td>i3.2xlarge</td>
<td>8</td>
<td>635</td>
</tr>
<tr>
<td>10.0 SE/EE</td>
<td>OWL2-RL</td>
<td>237,802,643</td>
<td>752,341,659</td>
<td>i3.4xlarge</td>
<td>16</td>
<td>607</td>
</tr>
</tbody>
</table>

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.
1.6.1.2 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.

<table>
<thead>
<tr>
<th>Server instance</th>
<th>Price</th>
<th>Disk</th>
<th>Concurrent read agents</th>
<th>Read query mixes per second</th>
<th>Concurrent write agents</th>
<th>Write per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>c5a.4xlarge</td>
<td>$0.616</td>
<td>EBS (5K IOPS)</td>
<td>0</td>
<td>•</td>
<td>4</td>
<td>20.46</td>
</tr>
<tr>
<td>i3.4xlarge</td>
<td>$1.248</td>
<td>local NVMe SSD</td>
<td>0</td>
<td>•</td>
<td>4</td>
<td>24.98</td>
</tr>
<tr>
<td>c5ad.4xlarge</td>
<td>$0.768</td>
<td>local NVMe SSD</td>
<td>0</td>
<td>•</td>
<td>4</td>
<td>32.73</td>
</tr>
<tr>
<td>c5a.4xlarge</td>
<td>$0.616</td>
<td>EBS (5K IOPS)</td>
<td>16</td>
<td>41.44</td>
<td>0</td>
<td>•</td>
</tr>
<tr>
<td>i3.4xlarge</td>
<td>$1.248</td>
<td>local NVMe SSD</td>
<td>16</td>
<td>99.11</td>
<td>0</td>
<td>•</td>
</tr>
<tr>
<td>c5ad.4xlarge</td>
<td>$0.768</td>
<td>local NVMe SSD</td>
<td>16</td>
<td>144.78</td>
<td>0</td>
<td>•</td>
</tr>
<tr>
<td>c5a.4xlarge</td>
<td>$0.616</td>
<td>EBS (5K IOPS)</td>
<td>8</td>
<td>30.61</td>
<td>4</td>
<td>6.50</td>
</tr>
<tr>
<td>i3.4xlarge</td>
<td>$1.248</td>
<td>local NVMe SSD</td>
<td>8</td>
<td>62.64</td>
<td>4</td>
<td>17.26</td>
</tr>
<tr>
<td>c5ad.4xlarge</td>
<td>$0.768</td>
<td>local NVMe SSD</td>
<td>8</td>
<td>71.33</td>
<td>4</td>
<td>21.69</td>
</tr>
<tr>
<td>c5a.4xlarge</td>
<td>$0.616</td>
<td>EBS (5K IOPS)</td>
<td>12</td>
<td>30.31</td>
<td>4</td>
<td>4.09</td>
</tr>
<tr>
<td>i3.4xlarge</td>
<td>$1.248</td>
<td>local NVMe SSD</td>
<td>12</td>
<td>78.95</td>
<td>4</td>
<td>11.94</td>
</tr>
<tr>
<td>c5ad.4xlarge</td>
<td>$0.768</td>
<td>local NVMe SSD</td>
<td>12</td>
<td>102.33</td>
<td>4</td>
<td>17.09</td>
</tr>
<tr>
<td>c5a.4xlarge</td>
<td>$0.616</td>
<td>EBS (5K IOPS)</td>
<td>16</td>
<td>24.44</td>
<td>4</td>
<td>2.94</td>
</tr>
<tr>
<td>i3.4xlarge</td>
<td>$1.248</td>
<td>local NVMe SSD</td>
<td>16</td>
<td>92.54</td>
<td>4</td>
<td>3.60</td>
</tr>
<tr>
<td>c5ad.4xlarge</td>
<td>$0.768</td>
<td>local NVMe SSD</td>
<td>16</td>
<td>117.56</td>
<td>4</td>
<td>10.23</td>
</tr>
</tbody>
</table>

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.
1.6.2 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 - c5ad.4xlarge, local NVMe SSD with GraphDB 10.0 EE, ruleset RDFS-Plus-optimized and excluded Query 5

<table>
<thead>
<tr>
<th>Threads</th>
<th>explore (query mixes per hour)</th>
<th>explore &amp; update (query mixes per hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77,861</td>
<td>25,441</td>
</tr>
<tr>
<td>2</td>
<td>145,719</td>
<td>50,153</td>
</tr>
<tr>
<td>4</td>
<td>256,025</td>
<td>60,972</td>
</tr>
<tr>
<td>8</td>
<td>411,748</td>
<td>61,557</td>
</tr>
<tr>
<td>12</td>
<td>423,748</td>
<td>60,371</td>
</tr>
<tr>
<td>16</td>
<td>470,485</td>
<td>60,770</td>
</tr>
</tbody>
</table>
2.1 Run GraphDB as a 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/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.

2.1.1 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 at http://localhost:7200/.

2.1.2 On MacOS

1. Download the GraphDB Desktop .dmg file.
2. Double-click it and get 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 at http://localhost:7200/.
2.1.3 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 at `http://localhost:7200`.

2.1.4 Configuring GraphDB

Once GraphDB Desktop is running, a small icon appears in the status bar/menu/tray area (varying depending on OS). It allows you to check whether the database is running, as well as to stop it or change the configuration settings. Additionally, an application window is also opened, where you can go to the GraphDB documentation, configure settings (such as the port on which the instance runs), and see all log files. You can hide the window from the `Hide window` button and reopen it by choosing `Show GraphDB window` from the menu of the aforementioned icon.
2.1.5 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: %APPDATA%/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:

```ini
[JavaOptions]
java-options=-Djpackage.app-version=10.0.0
java-options=-cp
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=-add-opens
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, it is recommended not to 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:

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

2.1.6 Stopping GraphDB

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

**Hint:** On some Linux systems, there is no support for status bar/menu/tray area. If you have hidden the GraphDB window, you can quit it by killing the process.

2.2 Run GraphDB as a Standalone Server

The default way of running 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 11 or greater.
2.2.1 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.

2.2.1.1 Options

The startup script supports the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d</code></td>
<td>daemonize (run in background), not available on Windows</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>run in server-only mode (no Workbench UI)</td>
</tr>
<tr>
<td><code>-p pidfile</code></td>
<td>write PID to pidfile</td>
</tr>
<tr>
<td><code>-h</code></td>
<td>print command line options</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>print GraphDB version, then exit</td>
</tr>
<tr>
<td><code>-Dprop</code></td>
<td>set Java system property</td>
</tr>
<tr>
<td><code>-Xprop</code></td>
<td>set non-standard Java system property</td>
</tr>
</tbody>
</table>

**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.

2.2.2 Configuring GraphDB

2.2.2.1 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:

```bash
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 as follows: command line supplied arguments > `GDB_JAVA_OPTS` > config file.
The GraphDB home directory

The GraphDB home directory 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.

2.2.2.2 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 setting one of the following environment variables:
- `GDB_HEAP_SIZE`: environment variable to set both the minimum and the maximum heap size (recommended);
- `GDB_MIN_MEM`: environment variable to set only the minimum heap size;
- `GDB_MAX_MEM`: environment variable to set 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 as follows: command line supplied arguments > `GDB_JAVA_OPTS` > `GDB_HEAP_SIZE` > `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, the most optimal performance can be achieved with either G1 GC or ParallelGC.

2.2.3 Stopping the database

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.
2.3 Set up Your License

GraphDB is available in three different editions: Free, Standard Edition (SE), and Enterprise Edition (EE).

The Free edition is free to use and does not require a license. This is the default mode in which GraphDB will start. However, it is not open-source.

SE and EE are RDBMS-like commercial licenses on a per-server-CPU basis. They are neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact graphdb-info@ontotext.com.

When installing GraphDB, the SE/EE license file can be set through the GraphDB Workbench or programmatically.

To do that, follow the steps:

1. Add, view, or update your license from Setup  Licenses  Set new license.

   Current license for this location

   GraphDB Free Edition

   Licensed to: Freeware  Valid until: Perpetual  Number of cores: 1  Maintenance date: Perpetual

   Set new license

   You do not have a valid license for GraphDB SE/EE and GraphDB works in Free mode. Set a valid SE/EE license to get access to all SE/EE features.

   From here, you can also Revert to Free license. If you do so, GraphDB will ask you to confirm.

   Confirm operation

   Reverting to a GraphDB Free license will make SE and EE features unusable. Are you sure?

   Cancel  Yes

2. Select the license file and register it.

   Register GraphDB License

   Click here to upload your license file

   or copy your license code below

   You can also copy and paste it in the text area.
3. Validate your license.

Are your license details correct?

GraphDB Enterprise Edition

<table>
<thead>
<tr>
<th>Licensed to</th>
<th>Valid until</th>
<th>Number of cores</th>
<th>Maintenance date</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONTOTEXT_INTERNAL</td>
<td>2022-12-31</td>
<td>Unlimited</td>
<td>2022-12-31</td>
</tr>
</tbody>
</table>

Commercial use for production environments

4. After completing these steps, you will be able to view your license details.

2.4 Create a Repository

Now let’s create your first repository.

**Hint:** When started, GraphDB creates the `GraphDB-HOME/data` directory to store local repository data. To change the directory, see Configuring GraphDB Data Directory.

1. Go to Setup → Repositories.
2. Click Create new repository.
3. Select GraphDB repository.

For Repository ID, enter my_repo and leave all other optional configuration settings at their default values.

**Tip:** For repositories with over several tens of millions of statements, see Configuring a Repository.

5. Click the Connect button to set the newly created repository as the repository for this location.
6. Use the pin to select it as the default repository.

Tip: You can also use cURL command to perform basic location and repository management through the GraphDB REST API.

2.5 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.

2.5.1 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), from a SPARQL construct query directly, or by typing or pasting the RDF data in a text area.

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.

### 2.5.2 Load data through SPARQL or 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 URL to clipboard](http://graphdb1.example.com:7200/repositories/my_repos)

   • Copy the repository URL.

2. Go to the folder where your local data files are.

3. Execute the script:
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.

### 2.5.3 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.

### 2.6 Explore Your Data and Class Relationships

#### 2.6.1 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.

Click on the settings icon on the top right for advanced graph settings. Control number of links, types, and predicates to hide and show.

A side panel opens with the available settings:

- **Graph Settings**
  - Maximum Links to Show
  - Preferred languages
  - Show Predicate Labels

**Types**
- Preferred types
- Show Preferred Types Only

**Predicates**
- Ignored types
- Add ignored type

**Settings**
- Reset
- Save
2.6.2 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.

Create visual graph config

- **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.
2.6.3 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.

**Class hierarchy**

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.

Sweet

• 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.
2.6.4 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.

### 2.6.5 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 the 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.

### 2.7 Query Your Data

#### 2.7.1 Query data through the Workbench

**Hint:** 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 Additional resources section.

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.

```sparql
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?x ?Person where {

(continues on next page)
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 rank:hasRDFRank ?rank .
} ORDER by DESC (?rank)
```

5. Find all people who are mentioned together with their political parties.

2.7. Query Your Data
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.

```sql
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?Mentions where {
}
```

<table>
<thead>
<tr>
<th>Mentions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 New York City</td>
</tr>
<tr>
<td>2 Andy Warhol</td>
</tr>
<tr>
<td>3 Pablo Picasso</td>
</tr>
<tr>
<td>4 Salvador Dali</td>
</tr>
<tr>
<td>5 Twilight Time</td>
</tr>
<tr>
<td>6 Wango</td>
</tr>
<tr>
<td>7 Jeff Koons</td>
</tr>
<tr>
<td>8 Dean Dubois</td>
</tr>
<tr>
<td>9 Prince</td>
</tr>
<tr>
<td>10 Bob Dylan</td>
</tr>
<tr>
<td>11 Museum of Art</td>
</tr>
<tr>
<td>12 Alberti Scagigetti</td>
</tr>
<tr>
<td>13 Mark Rothkov</td>
</tr>
<tr>
<td>14 Palace of Versailles</td>
</tr>
<tr>
<td>15 Hong Kong</td>
</tr>
<tr>
<td>16 Copenhagen</td>
</tr>
<tr>
<td>17 Tokyo</td>
</tr>
<tr>
<td>18 Paris</td>
</tr>
<tr>
<td>19 Berlin</td>
</tr>
<tr>
<td>20 Museum of Art</td>
</tr>
<tr>
<td>21 London</td>
</tr>
</tbody>
</table>

7. Find everything available about Marlon Brando in the database.

```sql
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?personLabel ?partyLabel where {
    ?mention pub-old:hasInstance ?person .
    ?person pub:memberOfPoliticalParty ?party .
    ?party pub:hasValue ?value .
}
```
8. Find all documents that mention members of the Democratic Party and the names of these people.

```sparql
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?document ?personLabel where {
  ?mention pub-old:hasInstance ?person .
  ?person pub:memberOfPoliticalParty ?party .
  ?party pub:hasValue ?value .
}
```

9. Find when these people were born and died.

```sparql
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?person ?personLabel ?dateOfBirth ?dateOfDeath where {
  OPTIONAL {
    ?person pub:dateOfBirth / pub:hasValue ?dateOfBirth .
  }
  OPTIONAL {
    ?person pub:dateOfDeath / pub:hasValue ?dateOfDeath .
  }
}
```
2.7.2 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%0D%0A+  %3Fp%0D%0A+  %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.
2.8 Additional Resources

SPARQL, OWL, and RDF:
- RDF: http://www.w3.org/TR/rdf11-concepts/
- RDFS: http://www.w3.org/TR/rdf-schema/
- SPARQL Overview: http://www.w3.org/TR/sparql11-overview/
- SPARQL Query: http://www.w3.org/TR/sparql11-query/
- SPARQL Update: http://www.w3.org/TR/sparql11-update
3.1 Requirements

3.1.1 Minimum requirements

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

- 2 gigabytes of memory
- 2 gigabytes of disk space
- Java SE Development Kit 11 to 16 (not required for GraphDB Free desktop installation)

**Warning:** All GraphDB indexes are optimized for hard disks with very low seek time. Our team highly recommend using only SSD partition for persisting repository images.

3.1.2 Hardware sizing

The best approach for correctly sizing the hardware resources is to estimate the number of explicit statements. Statistically, an average dataset has 3:1 statements to unique RDF resources. 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:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>are the planned number of explicit statements.</td>
</tr>
<tr>
<td>Unique resources</td>
<td>are the expected number of unique RDF resources (IRIs, blank nodes, literals, RDF-star [formerly RDF*] embedded triples).</td>
</tr>
<tr>
<td>Java heap (minimal)</td>
<td>is the minimal recommend JVM heap required to operate the database controlled by -Xmx parameter.</td>
</tr>
<tr>
<td>Java heap (optimal)</td>
<td>is the recommended JVM heap required to operate a database controlled by -Xmx parameter.</td>
</tr>
<tr>
<td>Off heap</td>
<td>is the database memory footprint (outside of the JVM heap) required to initialize the database.</td>
</tr>
<tr>
<td>OS</td>
<td>is the recommended minimal space reserved for the operating system.</td>
</tr>
<tr>
<td>Total</td>
<td>is the RAM required for the hardware configuration.</td>
</tr>
<tr>
<td>Repository image</td>
<td>is the expected size on disk. For repositories with inference use the total number of explicit + implicit statements.</td>
</tr>
</tbody>
</table>


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 half 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/writing data at once. The optimal buffer size depends on the hardware used, as reducing it would cause more write/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:

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Retained heap per repository</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>During prolonged import</td>
</tr>
<tr>
<td>Default</td>
<td>≥800MB</td>
</tr>
<tr>
<td>+ Reduced global cache (2GB)</td>
<td>670MB</td>
</tr>
<tr>
<td>+ Reduced buffer size*</td>
<td>570-620MB</td>
</tr>
<tr>
<td>+ Reduced inference pool size*</td>
<td>370-550MB</td>
</tr>
<tr>
<td>Serial import**</td>
<td>210-280MB</td>
</tr>
</tbody>
</table>

* Depends on the number of available CPU cores 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.

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

GraphDB is available in three different editions: Free, Standard Edition (SE), and Enterprise Edition (EE).

The Free edition is free to use and does not require a license. This is the default mode in which GraphDB will start. However, it is not open-source.

SE and EE are RDBMS-like commercial licenses on a per-server-CPU basis. They are neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact graphdb-info@ontotext.com.

When installing GraphDB, the SE/EE license file can be set through the GraphDB Workbench or programmatically.

3.2 Running 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. This is the easiest and fastest way to start using the GraphDB database.

- Run GraphDB as a desktop installation

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 as a standalone server

Run GraphDB in a Docker container - If you are into Docker and containers, we provide ready-to-use images for Docker. See more at https://github.com/Ontotext-AD/graphdb-docker.

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 at https://github.com/Ontotext-AD/graphdb-helm.

Note: The GraphDB Helm charts are an experimental feature under active development. More features will be added gradually and independently of GraphDB releases.

3.3 Configuring GraphDB

GraphDB 9.x relies on several main directories for configuration, logging, and data.

3.3.1 Directories

3.3.1.1 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.
GraphDB Documentation, Release 10.0.0

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.

Config directory

The GraphDB config directory defines where GraphDB looks for user-definable configuration. The config directory 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.

Important: 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`.

### 3.3.1.2 Checking the configured directories

When GraphDB starts, it logs the actual value for each of the above directories, e.g.:

<table>
<thead>
<tr>
<th>Directory Type</th>
<th>Directory Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home directory</td>
<td>/opt/test/graphdb-se-9.x.x</td>
</tr>
<tr>
<td>Config directory</td>
<td>/opt/test/graphdb-se-9.x.x/conf</td>
</tr>
<tr>
<td>Data directory</td>
<td>/opt/test/graphdb-se-9.x.x/data</td>
</tr>
<tr>
<td>Work directory</td>
<td>/opt/test/graphdb-se-9.x.x/work</td>
</tr>
<tr>
<td>Logs directory</td>
<td>/opt/test/graphdb-se-9.x.x/logs</td>
</tr>
</tbody>
</table>

### 3.3.2 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.

#### 3.3.2.1 Configuration properties

The properties are of four types and are detailed below.

**General properties**

The general properties define some basic configuration values that are shared with all GraphDB components and types of installation:
GraphDB Documentation, Release 10.0.0

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphdb.home</td>
<td>Defines the GraphDB home directory</td>
</tr>
<tr>
<td>graphdb.home.data</td>
<td>Defines the GraphDB data directory</td>
</tr>
<tr>
<td>graphdb.home.conf</td>
<td>(only as a system property) Defines the GraphDB conf directory</td>
</tr>
<tr>
<td>graphdb.home.work</td>
<td>Defines the GraphDB work directory</td>
</tr>
<tr>
<td>graphdb.home.logs</td>
<td>Defines the GraphDB logs directory</td>
</tr>
<tr>
<td>graphdb.dist</td>
<td>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).</td>
</tr>
<tr>
<td>graphdb.workbench.home</td>
<td>The place where the source for GraphDB Workbench is located</td>
</tr>
<tr>
<td>graphdb.license.file</td>
<td>Sets a custom path to the license file to use</td>
</tr>
<tr>
<td>graphdb.page.cache.size</td>
<td>The amount of memory to be taken by the page cache</td>
</tr>
<tr>
<td>graphdb.pidfile</td>
<td>The full path to the file where the GraphDB process ID is stored</td>
</tr>
<tr>
<td>graphdb.foreground</td>
<td>Tells GraphDB not to close stdout/stderr, but the user can choose whether to daemonize or not</td>
</tr>
<tr>
<td>graphdb.heapdump.enable</td>
<td>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.</td>
</tr>
<tr>
<td>graphdb.heapdump.path</td>
<td>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.</td>
</tr>
</tbody>
</table>

URL properties

**Hint:** 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphdb.vhosts</code></td>
<td>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 <code>/</code>, for example <code>http://example.com/rdf/</code>.</td>
</tr>
<tr>
<td><code>graphdb.external-url</code></td>
<td>Sets the canonical external URL. This property implies <code>graphdb.vhosts</code>. If you have provided an explicit value for both <code>graphdb.vhosts</code> and <code>graphdb.external-url</code>, then the URL specified for <code>graphdb.external-url</code> must be one of the URLs in the value for <code>graphdb.vhosts</code>. 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 <code>graphdb.vhosts</code>, as it will let GraphDB know that the canonical external URL is the one as seen through the proxy.</td>
</tr>
<tr>
<td><code>graphdb.hostname</code></td>
<td>Overrides the hostname reported by the machine.</td>
</tr>
</tbody>
</table>

**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:

```plaintext
# Required and recommended
graphdb.external-url = http://example.com/rdf/

# Non-recommended alternative to the above
#graphdb.vhosts = http://example.com/rdf/
```

**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. For more information, see Tomcat’s documentation.

Each property is composed of the prefix `graphdb.connector.` + the relevant Tomcat Connector attribute. 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.
### Table of Configuration Properties

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphdb.engine.entity-pool-implementation</td>
<td>Defines the Entity Pool implementation for the whole installation. Possible values are transactional or classic.</td>
<td>The default value is transactional. The transactional-simple implementation is not supported anymore.</td>
</tr>
<tr>
<td>graphdb.persistent.parallel.inferencers</td>
<td>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.</td>
<td>false</td>
</tr>
<tr>
<td>graphdb.engine.entity-validate</td>
<td>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.</td>
<td>true</td>
</tr>
</tbody>
</table>

**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.

#### 3.3.2.2 Configuring logging

GraphDB uses logback to configure logging. The default configuration is provided as `logback.xml` in the GraphDB `conf` directory.

### 3.4 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.

**Important:** Ontotext Refine is now developed as a separate product. All existing OntoRefine and RDF mapping functionalities remain available in it. See more in the Refine documentation.
Warning: The OntoRefine CLI is not available in Refine version 1.0 and will be available in 1.1. If you need this functionality, use OntoRefine with GraphDB 9.11 until Refine 1.1 is released. See OntoRefine CLI in the GraphDB 9.11 documentation.

3.4.1 Compatibility between the versions of GraphDB, Connectors, and third-party connectors

<table>
<thead>
<tr>
<th>GraphDB 10.0.0</th>
<th>RDF4J 4.0.2</th>
<th>Connectors Elasticsearch 7.16.3</th>
<th>Solr 8.11.1</th>
<th>Kafka 8.11.1</th>
<th>2.8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Introduced the <strong>new high-availability cluster</strong> where any node can be a leader or a follower (akin to the master and worker nodes in the old cluster). See the detailed migration procedure with a cluster below.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Implemented a new single repository type that replaces the existing Free, SE, and EE repositories: existing repositories will be automatically converted to the new type. If you have existing repository configuration templates outside a GraphDB installation, you need to convert them to the new type before using them with GraphDB 10.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Redesigned filtering mechanism in the connectors: you need to rewrite the filters and recreate the connectors. See <a href="#">Migrating connectors</a> below.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The GraphDB REST API has been refactored, with some of the changes including: moving the Import and SPARQL template controllers to a new base URL, the use of kebab-case for compound words in URLs, the removal of the <code>Header X-GraphDB-Password</code> from the Security management controller, and more. See <a href="#">Using the GraphDB REST API</a> for more information.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refactored remote locations – they cannot be activated any more but all repositories in remote locations are accessible via the Workbench.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OntoRefine has been removed from GraphDB and is now developed as a separate product - see note above.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you are upgrading from GraphDB 8.x or older, please upgrade to GraphDB 9.11 before you upgrade to GraphDB 10.0.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hint: See also the full release notes.

For older versions, please ask for support.
3.4.2 Migrating without a cluster

To migrate your GraphDB configuration and data, follow the steps below.

**Warning:** Keep in mind that after that, you cannot automatically revert to GraphDB 9.x.

1. Stop the GraphDB 9.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 9.x home directory (containing the **conf**, **data**, and **work** directories) can be used directly as the GraphDB 10.0 home directory.
   **Hint:** You can also copy the **conf**, **data**, and **work** directories from the GraphDB 9.x home directory to a new directory to use as the GraphDB 10.0 home directory. In this case, your GraphDB 9.x home directory is also the backup so you may skip the backup steps.

   The various directories are described in detail [here](#).

4. Start GraphDB 10.0.
5. If you use any GraphDB connectors, please follow the guidelines in Migrating connectors.

3.4.3 Migrating with a cluster

The cluster in GraphDB 10 is based on an entirely new approach and is not directly comparable or compatible with the cluster in GraphDB 9.x. See the High Availability Cluster Basics for more details on how the new GraphDB cluster operates.

The described procedures refer to the three recommended cluster topologies in the 9.x cluster: single master with three or more workers; two masters sharing workers, one of the masters is read-only; and multiple masters with dedicated workers. See more about 9.x cluster topologies.

3.4.3.1 Understand

You will need an existing GraphDB 9.x cluster in good condition before you start the migration. Data and configuration will be copied from two of the nodes:

- A **worker** node that is in sync with the master. This node will provide:
  - The data for each repository that is part of the GraphDB 9.x cluster.
  - Any repositories that are not part of the cluster, e.g., an Ontop repository created on the same instance as the worker repository. Typically, these are used via internal SPARQL federation in the cluster.

- A **master** node that will provide:
  - The user database containing users, credentials, and user settings.
  - Any repositories that are not part of the cluster, e.g., an Ontop repository created on the same instance as the master repository. Typically, these are used by connecting to the repository via HTTP – directly or via standard SPARQL federation.
  - The **graphdb.properties** file that contains all GraphDB configuration properties.
The instructions below assume your GraphDB 9.x setup has a single home directory that contains the `conf`, `data`, and `work` directories. If your setup uses explicitly configured separate directories for any of these, you need to adjust the instructions accordingly. The various directories are described in detail here.

**Important:** The cluster in GraphDB 10 is configured at the instance level, while the cluster in GraphDB 9.x is defined per repository. This means that every repository you migrate following the steps below will automatically become part of the cluster.

Once a cluster is created, it is not possible to have a repository that is not part of the cluster in GraphDB 10.

### 3.4.3.2 Prepare

In order to minimize downtime during the migration, you may want to keep the GraphDB 9.x cluster running in read-only mode while performing the migration.

To make a master read-only, go to **Setup ￿ Cluster**, click on the master node and enable the read-only setting:

Alternatively, you can reconfigure your application such that it does not do any writes during the migration.
3.4.3.3 Procedure

To migrate a cluster configuration from GraphDB version 9.x to the 10.0 cluster, please follow the steps outlined below.

**Warning:** The instructions are written in such a way that your existing GraphDB 9.x setup is preserved so you can abort the migration at any point and revert to your previous setup. Note that once you decide to go live with the migrated GraphDB 10 setup, there is no automatic way to revert that configuration to GraphDB 9.x.

1. First, choose a temporary GraphDB 10 home directory that will be used to copy files and directories and bootstrap all the nodes.

   **Hint:** All instructions below mean this directory when “temporary GraphDB 10 home directory” is mentioned.

2. Select one of the worker nodes that is in sync with the master.

3. Stop the GraphDB 9.x instance where the worker node is located – the rest of the GraphDB 9.x cluster will remain operational.

4. Locate the `data` directory within the GraphDB 9.x home directory of the worker node and copy it to the temporary GraphDB 10 home directory.
   - The `data/repositories` directory contains all repositories and their data.
   - If any repository is a master repository, delete it from the copy.

5. Select one of the master nodes.

6. Stop the GraphDB 9.x instance where the master node is located – you may want to point your application to another master or a worker repository so that read operations will continue to work during the migration.

7. Locate the `data` directory within the GraphDB 9.x home directory of the master node and copy it to the temporary GraphDB home directory.
   - The `data/repositories` directory contains all repositories and their data.
   - If any repository is a master repository, do not copy it.
   - If you have only master repositories on the master node you can skip this step.

8. Locate the `work` directory within the GraphDB 9.x home directory of the master node and copy it to the temporary GraphDB home directory.
   - On GraphDB 9.x, the `work` directory contains the user database.

   **Note:** After copying the `work` directory from the master to the new nodes, the old locations of the GraphDB 9.x cluster workers will be visible in the Workbench of the new nodes. We recommend deleting the old locations.

9. Locate the `conf` directory within the GraphDB 9.x home directory of the master node and copy it to the temporary GraphDB home directory.
   - The `conf` directory contains the `graphdb.properties` file.

10. Choose the number of nodes for the new cluster. Due to the nature of the Raft consensus algorithm on which the GraphDB 10 cluster is based, an **odd number of nodes** is recommended, e.g., three, five, or seven.

    As a rule of thumb, use as many nodes as the number of workers you had but add or remove a node to make the number odd. For example:
    - If you had three workers, use three nodes.
    - If you had six workers, use five or seven nodes.
11. Copy the temporary GraphDB 10 home directory to each node to serve as the GraphDB 10 on that node.

12. Edit the `graphdb.properties` file on each node to reflect any settings specific to that node, e.g., `graphdb.external-url` or SSL certificate properties but keep general properties, especially `graphdb.auth.token.secret` and any security-related properties identical on all nodes.
   - If necessary, consult the `graphdb.properties` file on that node from your GraphDB 9.x setup.
   - If the nodes are hosted on the same machine, edit the `graphdb.connector.port` property so that it is different for each node.
   - See also the notes on configuring networking properties related to the GraphDB 10 cluster.

13. Start GraphDB 10 on each node.
   - Make sure each node is up and has a valid EE license. If no license is applied, you will be able to create the cluster with all nodes in state **Follower** - no leader will be elected. However, if you attempt to run a query on any of them, their state will change to **Restricted**.

14. On any of the instances that you just created, go to Setup → Cluster in the Workbench and create the cluster group. See more information about the new Workbench user interface for creating, configuring, and accessing a cluster.
   - You can also create it via the Workbench **REST API**.

15. If you use any GraphDB connectors, please follow the guidelines in **Migrating connectors**.

### 3.4.3.4 Reverting the procedure

You can revert to your old setup by restarting the worker and master nodes that you stopped while performing the migration.

If you set your master to read-only, do not forget to set it back to write mode using the same Workbench interface that you used to make it read-only.

### 3.4.3.5 Example migration

Given the following GraphDB 9.x cluster setup consisting of two masters and three workers for each master, or a total of eight GraphDB instances:

**graphdb1.example.com**
- Master repository **master1**, the primary master repository
- Worker repository **mydata**, which is not part of any cluster

**graphdb2.example.com**
- Master repository **master2**, the secondary master repository

**graphdb3.example.com**
- Worker repository **worker1** connected to **master1**
- Ontop repository **sql1**

**graphdb4.example.com**
- Worker repository **worker2** connected to **master1**
- Ontop repository **sql1**

**graphdb5.example.com**
- Worker repository **worker3** connected to **master1**
- Ontop repository **sql1**
You choose the worker worker1 and the master master1 to perform the migration.

After completing the steps that copy files from the worker and the master, you should have a directory structure in the temporary GraphDB 10 home that looks like this:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data/repositories/worker1/</td>
<td>The worker repository copied from the worker node</td>
</tr>
<tr>
<td>data/repositories/sql1/</td>
<td>The Ontop repository copied from the worker node</td>
</tr>
<tr>
<td>data/repositories/mydata/</td>
<td>The non-clustered worker repository copied from the master node</td>
</tr>
<tr>
<td>conf/graphdb.properties</td>
<td>The GraphDB configuration file copied from the master node</td>
</tr>
<tr>
<td>work/workbench/settings.js</td>
<td>The GraphDB 9.x Workbench settings and user database copied from the master node</td>
</tr>
</tbody>
</table>

There may be other files in the data, conf, and work directories, e.g., conf/logback.xml, that are safe to have in the copy in order to preserve as much of the same configuration as possible.

Note, however, that you should NOT have the following directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data/repositories/master1/</td>
<td>The master repository from the master node should NOT be copied</td>
</tr>
</tbody>
</table>

Since you have six workers in the GraphDB 9.x cluster, it makes sense to choose five (the number of workers minus one to make the number odd) nodes for the GraphDB 10.0 cluster.

If you proceed with the migration, your cluster will contain three repositories that are part of the same cluster:

<table>
<thead>
<tr>
<th>Repository ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>worker1</td>
<td>Migrated GraphDB repository – note it uses the repository ID from the worker node you used to copy the files from</td>
</tr>
<tr>
<td>sql1</td>
<td>Migrated Ontop repository</td>
</tr>
<tr>
<td>mydata</td>
<td>Migrated GraphDB repository that previously was not part of any cluster</td>
</tr>
</tbody>
</table>
3.4.3.6 Configuring external cluster proxy

See how to configure the external GraphDB 10.0 cluster proxy here.

3.4.4 Migrating connectors

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 connector definitions do not include an entity filter, you can simply repair them.

If your connector definitions do include an entity filter, you need to rewrite the filter using the new filter options.

See the migration steps from GraphDB 9.x for Lucene, Solr, Elasticsearch, and Kafka.

3.4.5 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"
}
```

3.4.6 Migrating Helm charts

From version 9.8 onwards, GraphDB Enterprise Edition can be deployed with open-source Helm charts. See how to migrate them to GraphDB 10.0 here.

3.5 GraphDB Command Line Tools

The GraphDB distribution includes a number of command line tools located in the bin directory. Their file extensions are .sh or empty for Linux/Unix, and .cmd for Windows. They are described below in more detail.

3.5.1 console

This is an interactive console based on the RDF4J console.

Usage: start-console [OPTION] [repositoryID].

Use --help to see the available options, which are:
### 3.5.2 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.


The available options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;graphdb-pid&gt;</code></td>
<td>(Required) The process ID of a running GraphDB instance.</td>
</tr>
<tr>
<td><code>&lt;output-file&gt;</code></td>
<td>(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 <code>graphdb-server-report.zip</code> in the current directory.</td>
</tr>
</tbody>
</table>

### 3.5.3 graphdb

The `graphdb` command line tool starts the database. See more about its command line options here.

### 3.5.4 importrdf

The `importrdf` tool is used for offline loading of datasets. See more about it here.

### 3.5.5 rdfvalidator

Used for validating RDF files.

Usage: `rdfvalidator <input-folder-or-file-with-rdf-files>`.

### 3.5.6 reification-convert

This tool converts standard RDF reification to RDF-star. The output file must be an RDF-star format.


Available options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--relaxed</code></td>
<td>Enables relaxed mode where x a rdf:Statement is not required.</td>
</tr>
</tbody>
</table>
3.5.7 rule-compiler

Usage: rule-compiler <rules.pie> <java-class-name> <output-class-file> [partial].

Available options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;rules.pie&gt;</td>
<td>The name of the rule .pie file</td>
</tr>
<tr>
<td>&lt;java-class-name&gt;</td>
<td>The name of the Java class</td>
</tr>
<tr>
<td>&lt;output-class-file&gt;</td>
<td>The output file name</td>
</tr>
<tr>
<td>[partial]</td>
<td>(Optional)</td>
</tr>
</tbody>
</table>

3.5.8 storage-tool

The storage-tool is used for scanning and repairing a GraphDB repository. See more about its command line options here.

3.6 Distribution Package

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

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

After the first successful database run, the following directories will be generated, unless their default value is explicitly changed in conf/graphdb.properties:

<table>
<thead>
<tr>
<th>Default path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data/</td>
<td>Location of the repository data</td>
</tr>
<tr>
<td>logs/</td>
<td>Location for storing all database log files</td>
</tr>
<tr>
<td>work/</td>
<td>Work directory with non-user-editable configurations</td>
</tr>
</tbody>
</table>

3.7 Using Maven Artifacts

Part of GraphDB’s Maven repository is open and allows downloading GraphDB Maven artifacts without credentials.

Note: You still need to obtain a license from our Sales team, as the artifacts do not provide one.
3.7.1 Public Maven repository

To browse and search the public GraphDB’s Maven repository, use our Nexus.

For the Gradle build script:

```groovy
repositories {
    maven {
        url "https://maven.ontotext.com/repository/owlim-releases"
    }
}
```

For the Maven .POM file:

```xml
<repositories>
    <repository>
        <id>ontotext-public</id>
        <url>https://maven.ontotext.com/repository/owlim-releases</url>
    </repository>
</repositories>
```

3.7.2 Distribution

To use the distribution for some automation or to run integration tests in embedded Tomcat, get the .zip artifacts with the following snippet:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <id>copy</id>
                    <phase>package</phase>
                    <goals>
                        <goal>copy</goal>
                    </goals>
                    <configuration>
                        <artifactItems>
                            <artifactItem>
                                <groupId>com.ontotext.graphdb</groupId>
                                <artifactId>graphdb</artifactId>
                                <version>${graphdb.version}</version>
                                <type>zip</type>
                                <classifier>dist</classifier>
                                <outputDirectory>target/graphdb-dist</outputDirectory>
                            </artifactItem>
                        </artifactItems>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```
3.7.3 GraphDB runtime .jar

To embed the database in your application or develop a plugin, you need the GraphDB runtime .jar. Here are the details for the runtime .jar artifact:

```xml
<dependency>
  <groupId>com.ontotext.graphdb</groupId>
  <artifactId>graphdb-runtime</artifactId>
  <version>${graphdb.version}</version>
  <!-- Temporary workaround for missing Ontop dependencies for Ontotext build of Ontop -->
  <exclusions>
    <exclusion>
      <groupId>it.unibz.inf.ontop</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

The com.ontotext.graphdb:graphdb-runtime artifact is also available from the Maven Central Repository.

3.7.4 GraphDB Client API .jar

The GraphDB Client API is an extension of RDF4J's HTTP repository and provides some GraphDB extensions and smart GraphDB cluster support. Here are the details for the .jar artifact:

```xml
<dependency>
  <groupId>com.ontotext.graphdb</groupId>
  <artifactId>graphdb-client-api</artifactId>
  <version>${graphdb.version}</version>
</dependency>
```

The com.ontotext.graphdb:graphdb-client-api artifact is also available from the Maven Central Repository.
4.1 Workbench User Interface

The Workbench is the web-based administration interface to GraphDB. It lets you administrate 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 to all functionalities - Import, Explore, SPARQL, Monitor, Setup, and Help. The work area shows the tasks associated with the selected functionality. The home page provides easy access to some of the actions in the Workbench such as creating a repository, attaching a location, finding a resource, querying your data, etc. At the bottom of the page, you can see the license details, and in the footer - the versions of the various GraphDB components.
4.1.1 Functionalities

<table>
<thead>
<tr>
<th>Navigation Tab</th>
<th>Functionality Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Import</strong></td>
<td>• 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.</td>
</tr>
</tbody>
</table>
| **Explore**     | • *Graphs overview* → See a list of the default graph and all named graphs in GraphDB. Use it to inspect the statements in each graph, export the graph, or clear its 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 and it gets the color of the class with more incoming links. The links can be in both directions.  
|                 | • *Visual graph* → Explore your data graph in a visual way. 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 as well.  
|                 | • *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. |
| **Monitor**     | • *Queries and Updates* → Monitor all running queries or updates in GraphDB. Any query or update can be killed by pressing the Abort button.  
|                 | • *Resources* → Monitor the usage of various system resources, such as memory and CPU, for the currently active location. |

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Navigation Tab</th>
<th>Functionality Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Setup</strong></td>
<td></td>
</tr>
<tr>
<td>• <strong>Repositories</strong> –&gt; 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.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Users and Access</strong> –&gt; 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 functionality.</td>
<td></td>
</tr>
<tr>
<td>• <strong>My Settings</strong> –&gt; Configure the default behavior of the Workbench.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Connectors</strong> –&gt; Create and manage GraphDB Connector instances.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Cluster</strong> –&gt; 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.</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> This feature requires a GraphDB Enterprise license.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Namespaces</strong> –&gt; View and manipulate the RDF namespaces for the active repository. You need a write permission to add or delete namespaces.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Autocomplete</strong> –&gt; Enable/disable the autocomplete index and check its status. It is used for automatic completion of URIs in the SPARQL editor and the View Resource page.</td>
<td></td>
</tr>
<tr>
<td>• <strong>RDF Rank</strong> –&gt; 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.</td>
<td></td>
</tr>
<tr>
<td>• <strong>JDBC</strong> –&gt; Configure the JDBC driver to allow SQL access to repository data.</td>
<td></td>
</tr>
<tr>
<td>• <strong>SPARQL Templates</strong> –&gt; Create and store predefined SPARQL templates for future updates of repository data.</td>
<td></td>
</tr>
<tr>
<td>• <strong>License</strong> –&gt; View the details of your current GraphDB license and set or revert to a different one.</td>
<td></td>
</tr>
<tr>
<td><strong>Help</strong></td>
<td></td>
</tr>
<tr>
<td>• <strong>REST API</strong> –&gt; REST API documentation of all available public RESTful endpoints together with an interactive interface for executing requests.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Documentation</strong> –&gt; Link to the GraphDB public documentation.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Developer Hub</strong> –&gt; Link to the GraphDB dev hub - a hands-on compendium to the GraphDB documentation that gives practical advice and tips on accomplishing real-world tasks.</td>
<td></td>
</tr>
<tr>
<td>• <strong>Support</strong> –&gt; Link to the GraphDB support page.</td>
<td></td>
</tr>
<tr>
<td>• <strong>System information</strong> –&gt; See the configuration values of the JVM running the GraphDB Workbench: Application Info, JVM Arguments, and Workbench Configuration properties. You can also generate a detailed server report file that you can use to hunt down issues.</td>
<td></td>
</tr>
</tbody>
</table>
4.1.2 Configuration 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`):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphdb.workbench.cors.enable</code></td>
<td>Enables cross-origin resource sharing.</td>
<td>false</td>
</tr>
<tr>
<td><code>graphdb.workbench.cors.origin</code></td>
<td>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.</td>
<td>*</td>
</tr>
<tr>
<td><code>graphdb.workbench.cors.expose-headers</code></td>
<td>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: <code>graphdb.workbench.cors.enable=true</code> <code>graphdb.workbench.cors.origin=*</code> <code>graphdb.workbench.cors.expose-headers=&quot;content,location&quot;</code></td>
<td>If no value is set, only the CORS-safelisted request headers will be exposed.</td>
</tr>
<tr>
<td><code>graphdb.workbench.maxConnections</code></td>
<td>Sets the maximum number of concurrent connections to a remote GraphDB instance.</td>
<td>200</td>
</tr>
<tr>
<td><code>graphdb.workbench.datadir</code></td>
<td>Sets the directory where the workbench persistence data will be stored.</td>
<td><code>${user.home}/.graphdb-workbench/</code></td>
</tr>
<tr>
<td><code>graphdb.workbench.importDirectory</code></td>
<td>Changes the location of the file import folder.</td>
<td><code>${user.home}/graphdb-import/</code></td>
</tr>
<tr>
<td><code>graphdb.workbench.maxUploadSize</code></td>
<td>Sets the maximum upload size for importing local files. The value must be in bytes.</td>
<td>200 megabytes</td>
</tr>
</tbody>
</table>

4.2 Setting up Licenses

GraphDB is available in three different editions: Free, Standard Edition (SE), and Enterprise Edition (EE).

The Free edition is free to use and does not require a license. This is the default mode in which GraphDB will start. However, it is not open-source.

SE and EE are RDBMS-like commercial licenses on a per-server-CPU basis. They are neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact `graphdb-info@ontotext.com`.

When installing GraphDB, the SE/EE license file can be set through the GraphDB Workbench or programmatically.
4.2.1 Setting up licenses through the Workbench

To do that, follow the steps:

1. Add, view, or update your license from Setup → Licenses → Set new license.

   Current license for this location

   GraphDB Free Edition

   Licensed to: Freeware
   Valid until: Perpetual
   Number of cores: 1
   Maintenance date: Perpetual

   Set new license

   You do not have a valid license for GraphDB SE/EE and GraphDB works in Free mode. Set a valid SE/EE license to get access to all SE/EE features.

   From here, you can also Revert to Free license. If you do so, GraphDB will ask you to confirm.

2. Select the license file and register it.

   You can also copy and paste it in the text area.
3. Validate your license.

Are your license details correct?

GraphDB Enterprise Edition
Licensed to ONTOTEXT_INTERNAL
Valid until 2022-12-31
Number of cores Unlimited
Maintenance date 2022-12-31

4. After completing these steps, you will be able to view your license details.

4.2.2 Setting up licenses through a file

GraphDB will look for a `graphdb.license` file in the GraphDB configuration directory (by default `conf` under `GraphDB-HOME`). To install a license file there, copy the license file as `graphdb.license`.

4.2.2.1 Custom file path property

You can use the configuration property `graphdb.license.file` to provide a custom path for the license file, for example:

```
grapdab.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 `config` directory or a custom path, you will not be able to change the license through the GraphDB Workbench.

4.2.3 Order of preference

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 configuration directory;
- A license set through the GraphDB Workbench.

4.3 Creating Locations

Locations represent individual GraphDB servers where the repository data is stored. They can be local (a directory on the disk) or remote (an endpoint URL), and can be attached, edited, and detached.

Remote locations are useful when you want to quickly connect another GraphDB instance and execute operations on it. They are also a key component of cluster creation.

The local location is the one that is visible to the RDF4J REST API, while the remote locations are visible and can be used only through the Workbench. All of the locations are always active at the same time.
4.3.1 Connect to a remote location

To connect to a remote location:

1. Start a browser and go to the Workbench web application using a URL of the form http://localhost:7200, substituting localhost and the 7200 port number as appropriate.

2. Go to Setup  › Repositories.

3. Click the Attach remote location button and and enter the URL of the remote GraphDB instance, for example http://localhost:7202.

4. In terms of authentication methods to the remote location, GraphDB offers three options:

   a. **None**: The security of the remote location is disabled, and no authentication is needed.

   ![Attach a remote GraphDB instance](image)

   b. **Basic authentication**: The security of the remote location has basic authentication enabled (default setting). Requires a username and a password.

   ![Attach a remote GraphDB instance](image)

   c. **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.

   ![Attach a remote GraphDB instance](image)

   **Hint**: Signature authentication is the recommended method for a cluster environment, as both require the same authentication settings.

   ![Attach a remote GraphDB instance](image)

5. After the location has been created, it will appear right below the local one.
4.3.1.1 Change location settings

The location setting for sending anonymous statistics to Ontotext 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 *Edit common settings for these repositories*.

The following settings dialog will appear:

- **Send anonymous usage statistics to Ontotext**
- **Why should you send us your statistics?**
  - We aim to provide better products that suit the needs of our users and customers. The anonymous statistics help us understand those needs better and focus our efforts.
- **What do the statistics include?**
  - For each repository we gather:
    - *Absolute values*
      - GraphDB edition and version
      - Number of explicit and implicit triples
      - Number of entities
      - Number of predicates
      - Size of repository on disk
      - Whether a given plug-in is used, e.g., geo-spatial
      - Ruleset (custom rulesets are reported only as "custom")
      - DD type
    - *Aggregated values*
      - Number of queries per day
      - Average time per query per day
      - Number of updates per day
      - Average time per update per day

The data is sent every 24 hours over HTTP to a dedicated endpoint at http://statistics.graphdb.ontotext.com.

The data is encrypted with a 2048-bit RSA key.

The settings can be saved or cancelled.
4.3.1.2 View or update location license

Click the key icon to check the details of your current license.

**Hint:** Signature authentication is the recommended method for a cluster environment, as both require the same authentication settings.

**Note:** As mentioned above, all repositories in the local location are visible through the RDF4J REST API, including as a SPARQL endpoint.

**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. In case the certificate of the remote host is self-signed, you should add it to you JVM’s SSL TrustStore.

4.3.2 Configure a data location

Set the property `graphdb.home.data` in `conf/graphdb.properties`. If no property is set, the default repositories location will be in the `data` directory of the GraphDB distribution.

4.4 Creating a Repository

4.4.1 Create a repository

There are two ways for creating and managing repositories: either through the Workbench interface, or by using the RDF4J console.

4.4.1.1 Using the Workbench

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 or create it from a file by using the configuration template that can be found under `<configs/templates/>` in the GraphDB distribution.
2. Select GraphDB repository.

3. If you have attached remote locations to your GraphDB instance, specify 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.

4. Enter the Repository ID (e.g., my_repo) and leave all other optional configuration settings with their default values.

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

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

---

### Create a repository in a remote location

You can easily create a repository in any remote location attached to your GraphDB instance.

1. First, connect to the location. For this example, let’s connect `http://localhost:7202` (substitute `localhost` and the `7200` port number as appropriate).

2. Then, just like with local repositories, go to Setup > Repositories > Create new repository.

3. In the Location field, the two locations are now visible. Select `http://localhost:7202`.

4. Fill in the Repository ID and create the repo.

5. Both repositories are now created.

6. If you go to the `http://localhost:7202` location, you will see `remote_repo` created there.
4.4.1.2 Using the RDF4J console

**Note:** Use the `create` command to add new repositories to the location to which the console is connected. This command expects the name of the template that describes the repository’s configuration.

1. Run the RDF4J console application, which resides in the `/bin` folder:
   - Windows: `console.cmd`
   - Unix/Linux: `./console`
2. Connect to the GraphDB server instance using the command:
   ```
   ```
3. Create a repository using the command:
   ```
   create graphdb.
   ```
4. Fill in the values of the parameters in the console.
5. Exit the RDF4J console:
   ```
   quit.
   ```

4.4.2 Manage repositories

4.4.2.1 Connect a repository

Click the connect icon in the repository list.

Alternatively, 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 the respective repository will also display some basic information about it.

Note that when you are connected to a remote repository, its label in the top right corner indicates that:
4.4.2.2 Make it the default repository

Use the pin to select it as the default repository.

4.4.2.3 Edit a repository

To copy the repository URL, edit it, download the repository configuration as a Turtle file, restart it, or delete it, use the respective icons next to its name.

**Warning:** Once a repository is deleted, all data contained in it is irrevocably lost.

You can restart a repository without having to restart the entire GraphDB instance. There are two ways to do that:

- Click the restart icon as shown above. A warning will prompt you to confirm the action.
- Click the edit icon, which will open the repository configuration. At its bottom, 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.

4.5 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 the configuration template is and how it works, what data structures GraphDB supports, what configuration values are optimal for your setup, etc.
4.5.1 Plan a repository configuration

To plan your repository configuration, check out the following sections:

- **Hardware sizing**
- **Configuration parameters**
- **How the template works**
- **GraphDB data structures**
- **Configure Java heap memory**
- **Configure Entity pool memory**

4.5.2 Configure a repository through the GraphDB Workbench

To configure a new repository, complete its properties form.

![Create GraphDB repository form](image)

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

4.5.3 Edit a repository

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

1. Click the *Edit* icon next to a repository to edit it.
2. Restart GraphDB for the changes to take effect.
4.5.4 Configure a repository programmatically

**Tip:** GraphDB uses an RDF4J configuration template for configuring 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. For more information on how the configuration template works, see Repository configuration template - how it works.

To configure a new repository programmatically:

1. Fill in the .ttl configuration template that can be found in the /configs/templates folder of the GraphDB distribution. The parameters are described in the Configuration parameters section.

```turtle
# RDF4J 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 "gdb" ;
    rdfs:label "" ;
    rep:repositoryImpl [ rep:repositoryType "graphdb:SailRepository" ;
        sr:sailImpl [ sail:sailType "graphdb:Sail" ;
            graphdb:base-URL "http://example.org/" ;
            graphdb:defaultNS "" ;
            graphdb:entity-index-size "10000000" ;
            graphdb:entity-id-size "32" ;
            graphdb:imports "" ;
            graphdb:repository-type "file-repository" ;
            graphdb:ruleset "owl-horst-optimized" ;
            graphdb:storage-folder "storage" ;
            graphdb:enable-context-index "false" ;
            graphdb:enablePredicateList "true" ;
            graphdb:in-memory-literal-properties "true" ;
            graphdb:enable-literal-index "true" ;
            graphdb:check-for-inconsistencies "false" ;
            graphdb:disable-sameAs "true" ;
            graphdb:query-timeout "0" ;
            graphdb:query-limit-results "0" ;
            graphdb:throw-QueryEvaluationException-on-timeout "false" ;
            graphdb:read-only "false" ;
```

To configure a SHACL validation enabled repository programmatically, do the same as above, but with the added SHACL parameters:

```turtle
# RDF4J configuration template for a GraphDB repository with SHACL validation
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix support: "".

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
```

(continues on next page)
2. Use this command to create a repo from the config.ttl:

```
graphdb:base-URL "http://example.org/";
graphdb:check-for-inconsistencies "false";
graphdb:defaultNS "";
graphdb:disable-sameAs "true";
graphdb:enable-context-index "false";
graphdb:enable-literal-index "true";
graphdb:enablePredicateList "true";
graphdb:entity-id-size "32";
graphdb:entity-index-size "100000000000";
graphdb:imports "";
graphdb:in-memory-literal-properties "true";
graphdb:query-limit-results "0";
graphdb:query-timeout "0";
graphdb:read-only "false";
graphdb:repository-type "file-
repository";
graphdb:ruleset "rdfsplus-optimized";
graphdb:storage-folder "storage";
graphdb:throw-QueryEvaluationException-on-timeout "false";
```

4.5. Configuring a Repository 77
4.5.5 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 (e.g., unsupported inferred statements, missing inferred statements, or inferred statements that cannot be deleted).
<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>base-URL</td>
<td>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.</td>
<td>none</td>
</tr>
<tr>
<td>check-for-inconsistencies</td>
<td>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 is while committing a transaction, the whole transaction will be rolled back.</td>
<td>false</td>
</tr>
</tbody>
</table>
| defaultNS                      | Default namespaces corresponding to each imported schema file, separated by semicolon. The number of namespaces must be equal to the number of schema files from the imports parameter. Example: graphdb:defaultNS "http://www.w3.org/2002/07/owlt#:http://example.org/"  
  **Warning:** This parameter cannot be set via a command line argument. | <empty>                       |
| disable-sameAs                 | Enables or disables the owl:sameAs optimization.                                                                                                                                                                                                                                                                                           | true                          |
| enable-context-index           | Possible value: true, where GraphDB will build and use the context index.                                                                                                                                                                                                                                                               | false                         |
| enable-literal-index           | Enables or disables the storage. The literal index is always built as data is loaded/modified. This parameter only affects whether the index is used during query answering.                                                                                                           | true                          |
| 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.                                                                                                                                                       | true                          |
| entity-id-size                 | Defines the bit size of internal IDs used to index entities this parameter can be left at its default value. However, if using very large datasets containing over 2^{31} entities, set this parameter to 40.                                                                                                                                               | 32                            |

### 4.5. Configuring a Repository

Be aware that this can only be set when instantiating a new repository, and that converting an existing repository from 32 to 40-bit entity widths is not possible.
4.5.6 Configure GraphDB memory

4.5.6.1 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.

As a general rule, the `-Xmx` value should not exceed 2/3 of the system memory. This means that if you have a system with a total of 8 gigabytes RAM, where 1 gigabyte is used by the operating system, services, etc., and 1 gigabyte by the entity pool and the hash maps. As they are off-heap, the JVM that hosts the application using GraphDB should, ideally, have a maximum heap size of 6 gigabytes, and can be set using the JVM argument `-Xmx6g`.

4.5.6.2 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 current global cache implementation can be enabled by specifying: `-Dgraphdb.global.page.cache=true -Dgraphdb.page.cache.size=3G`.

If you do not specify `graphdb.page.cache.size` but only enable the global cache, it will take 50% of the `-Xmx` parameter.

**Note:** You do not have to change/edit your repository configurations. The new cache will be used when you upgrade to the new version.

4.5.6.3 Configure Entity pool memory

By default, all entity pool structures are residing off-heap, i.e., outside of the normal JVM heap. This way, you do not have to calculate the entity pool memory when giving the JVM max heap memory parameter to GraphDB. This means, however, that you need to leave some memory outside of the `-Xmx`.

To activate the old behavior, you can still enable on-heap allocation with `-Dgraphdb.epool.onheap=true`.

If you are concerned that the process will eat up 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).
4.5.6.4 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.

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th>Description</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total OS memory</td>
<td>Total physical system memory</td>
<td>16 GB</td>
</tr>
<tr>
<td>On-heap JVM (-Xmx) configuration</td>
<td>Maximum heap memory allocated by the JVM process</td>
<td>10 GB</td>
</tr>
<tr>
<td>graphdb.page.cache.size</td>
<td>Global single cache shared between all internal</td>
<td>5 GB</td>
</tr>
<tr>
<td></td>
<td>structures of all repositories (the default value is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50% of the heap size)</td>
<td></td>
</tr>
<tr>
<td>Remaining on-heap memory for query</td>
<td>Raw estimate of the memory for query execution; a</td>
<td>~4.5 GB</td>
</tr>
<tr>
<td>execution</td>
<td>higher value is required if many, long running</td>
<td></td>
</tr>
<tr>
<td></td>
<td>analytical queries are expected</td>
<td></td>
</tr>
<tr>
<td>entity-index-size (&quot;Entity index size&quot;)</td>
<td>Size of the initial entity pool hash table; the</td>
<td>150,000,000</td>
</tr>
<tr>
<td>stored off-heap by default</td>
<td>recommended value is equal to the total number of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unique entities</td>
<td></td>
</tr>
<tr>
<td>Memory footprint of the entity pool</td>
<td>Calculated from entity-index-size and total number</td>
<td>~2.5 GB</td>
</tr>
<tr>
<td>stored off-heap by default</td>
<td>of entities; this memory will be taken after the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>repository initialization</td>
<td></td>
</tr>
<tr>
<td>Remaining OS memory</td>
<td>Raw estimate of the memory left to the OS</td>
<td>~3.5 GB</td>
</tr>
</tbody>
</table>

4.5.6.5 Upper bounds for the memory consumed by the GraphDB process

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 -Ddefault.min.distinct.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.

4.5.7 Reconfigure a repository

Once a repository is created, it is possible to 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.

4.5.7.1 Using the 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.
4.5.7.2 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 on how to do it.

4.5.8 Rename a repository

4.5.8.1 Using the Workbench

Use the Workbench to change the repository ID. This will update all locations in the Workbench where the repository name is used.

4.6 Creating and Managing a Cluster

The below instructions will walk you through the steps for creating and monitoring a cluster group.

4.6.1 Prerequisites

You will need at least three GraphDB installations to create a fully functional cluster. Remember that the Raft algorithm recommends an odd number of nodes, so a cluster of five nodes is a good choice too.

All of the nodes must have the same security settings, and in particular the same shared token secret even when security is disabled.

For all GraphDB instances, set the following configuration property in the graphdb.properties file and change <my-shared-secret-key> to the desired secret:

```
graphdb.auth.token.secret = <my-shared-secret-key>
```

All of the nodes must have their networking configured correctly – the hostname reported by the OS must be resolvable to the correct IP address on each node that will participate in the cluster. In case your networking is not configured correctly or you are not sure, you can set the hostname for each node by putting graphdb.hostname = hostname.example.com into each graphdb.properties file, where hostname.example.com is the hostname for that GraphDB to use in the cluster. 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.

4.6.2 High availability deployment

A typical deployment scenario would be a deployment in cloud infrastructure with the ability to deploy GraphDB instances in different regions or zones so that if a region/zone fails, the GraphDB cluster will continue functioning without any issues for the end-user.

To achieve high availability, it is recommended to deploy GraphDB instances in different zones/regions while considering the need for a majority quorum in order to be able to accept INSERT/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.
4.6.2.1 Cluster group with three nodes

In a cluster with three nodes, we need at least two in order to be able 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.

4.6.2.2 Cluster group with five nodes

In a cluster with five nodes, we need three nodes for a quorum. If we have three available regions/zones, we can deploy:

- two instances in zone 1,
- two instances in zone 2,
- one instance in zone 3.

If any of the zones fail, we would still have at least three more GraphDB instances that will form a quorum.

4.6.3 Create cluster

A cluster can be created interactively from the Workbench or programatically via the REST API.

4.6.3.1 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.

   ![Cluster management](image)

   Click the icon to create a cluster group.

2. In the dialog that opens, you can see that the current GraphDB node is discovered automatically. Click Attach remote location and add the other four instances as remote instances:
This is essentially the same operation as when attaching a remote location. Clicking on Advanced settings opens an additional panel with settings that affect the entire cluster group but the defaults should be good for a start:

3. Once you have added all nodes (in this case graphdb2.example.com:7200, graphdb3.example.com:7200, graphdb4.example.com:7200, and graphdb5.example.com:7200, since graphdb1.example.com:7200 was discovered automatically and always has to be part of the cluster), click on each of them to include them in the cluster group:

4. Click OK.
5. At first, all nodes become followers (colored in blue). Then one of the nodes initiates election, after which for a brief moment, one node becomes a candidate (you may see it briefly flash in green), and finally a leader (colored in orange).

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.

All possible node and connection states are listed in the legend on the bottom left that you can toggle by clicking the question mark icon.
6. You can also add or remove nodes from the cluster group, as well as delete it.

See also Using a Cluster.

4.6.3.2 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 cURL.

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):

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 using graphdb.cluster.rpc.port = NNNN, where NNNN is the chosen port.

```
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"
   ]}
```

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": 1500,
   "electionRangeTimeout": 2000,
   "heartbeatInterval": 500,
   "messageSizeKB": 64,
   "verificationTimeout": 1500
}
```

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 configuration of the cluster group is invalid, the returned response code will be 400 Bad request.

412: Unreachable nodes or cluster already existing on a node
If at cluster config creation, 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"
  }
  ```

4.6.3.3 Creation parameters

The cluster group configuration has several properties that have sane default values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>electionMinTimeout</td>
<td>1500</td>
<td>The minimum wait time in milliseconds for a heartbeat from a leader.</td>
</tr>
<tr>
<td>electionRangeTimeout</td>
<td>2000</td>
<td>The variable portion of each waiting period in milliseconds for a heartbeat.</td>
</tr>
<tr>
<td>heartbeatInterval</td>
<td>500</td>
<td>The interval in milliseconds between each heartbeat that is sent to follower nodes by the leader.</td>
</tr>
<tr>
<td>verificationTimeout</td>
<td>1500</td>
<td>The amount of time in milliseconds a follower node would wait before attempting to verify the last committed entry when the first verification is unsuccessful.</td>
</tr>
<tr>
<td>messageSizeKB</td>
<td>64</td>
<td>The size of the data blocks transferred during data replication streaming through the RPC protocol.</td>
</tr>
</tbody>
</table>

4.6.4 Manage cluster membership

We can add and remove cluster nodes at runtime without having to stop the entire cluster group. This is achieved through total consensus between the nodes in the new configuration when making a change to the cluster membership.

When adding nodes, a total consensus means that all nodes, both the new and the old ones, have successfully appended the configuration.

If there is no majority of nodes responding to heartbeats, we can remove the non-responsive ones all at once. In this situation, a total consensus on the new configuration would be enough for this operation to be executed successfully.

It is recommended to remove fewer than 1/2 of the nodes from the current configuration.
4.6.4.1 Add nodes

Using the Workbench

New nodes can be added to the cluster group only from the leader. From Setup → Cluster → Add nodes, just like with node creation, attach the node’s HTTP address as a remote location and click OK.

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 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.

Only the leader can make cluster membership changes, so if a follower tries to add a node to the cluster group, again an error message will be returned.

The added node 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.

4.6.4.2 Remove nodes

Using the Workbench

Nodes can be removed from the cluster group only from the leader. From Setup → Cluster → Remove nodes.
Remove nodes from cluster

Select nodes to be removed from existing cluster

Cluster nodes list

http://graphdb1.example.com:7200
http://graphdb2.example.com:7200
http://graphdb3.example.com:7200
http://graphdb4.example.com:7200
http://graphdb5.example.com:7200

Click on the nodes that you want to remove and click OK.

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.

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, the cluster only on this node can be manually deleted from Setup  Cluster  Delete cluster.
4.6.5 Manage cluster configuration properties

You can view and manage the cluster configuration properties both from the Workbench and the REST API.

4.6.5.1 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 config properties and a list of its nodes.

![Cluster configuration](image)

**Cluster configuration**

- Election minimum timeout: 20,000
- Election range timeout: 10,000
- Heartbeat interval: 2,000
- Message size KB: 64
- Verification timeout: 1,500

**Nodes list**

- [http://graphdb1.example.com:7200](http://graphdb1.example.com:7200)
  - RPC address: graphdb1.example.com:7300
  - Node state: LEADER
- [http://graphdb2.example.com:7200](http://graphdb2.example.com:7200)
  - RPC address: graphdb2.example.com:7300
  - Node state: FOLLOWER
- [http://graphdb3.example.com:7200](http://graphdb3.example.com:7200)
  - RPC address: graphdb3.example.com:7300
  - Node state: FOLLOWER
- [http://graphdb4.example.com:7200](http://graphdb4.example.com:7200)
  - RPC address: graphdb4.example.com:7300
  - Node state: FOLLOWER
- [http://graphdb5.example.com:7200](http://graphdb5.example.com:7200)
  - RPC address: graphdb5.example.com:7300
  - Node state: FOLLOWER

To modify the config properties, click *Edit configuration*.

**Important:** Editing of these properties is only possible on the leader node.
4.6.5.2 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:

```bash
curl http://graphdb1.example.com:7200/rest/cluster/config
```

To check the cluster configuration, go to GET /rest/cluster/config and click Try it out.

**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 config properties, perform a PATCH request containing the parameters of the new config to the /rest/cluster/config endpoint:

```bash
curl -X PATCH -H 'Content-Type: application/json' \
'http://graphdb1.example.com:7200/rest/cluster/config' \ 
-d '{
"electionMinTimeout": 7300,
"electionRangeTimeout": 5000,
"heartbeatInterval": 2000,
"messageSizeKB": 64,
"verificationTimeout": 1500
}'
```

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 because we need a total consensus between the nodes, so if one of them cannot append the new config, all of them will reject it.

4.6.6 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 will display 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.

### 4.6.6.1 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 the following result:

```json
[
  {
    "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
  },
  {
    "address": "graphdb2.example.com:7300",
    "endpoint": "http://graphdb2.example.com:7200",
    "lastLogIndex": 0,
    "lastLogTerm": 0,
    "nodeState": "FOLLOWER",
    "syncStatus": {},
    "term": 2
  },
  {
    "address": "graphdb3.example.com:7300",
    "endpoint": "http://graphdb3.example.com:7200",
    "lastLogIndex": 0,
    "lastLogTerm": 0,
    "nodeState": "FOLLOWER",
    "syncStatus": {},
    "term": 2
  }
]
```

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.
4.6.6.2 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
}
```

For a follower node (is 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
}
```

4.6.7 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.

**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"
}
```

4.6.7.1 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:

![Cluster delete confirmation](image)

4.6.8 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 one the leader is and always serve all requests to the current leader.

**Note:** The external proxy does not require a GraphDB SE/EE license.
4.6.8.1 Start the external proxy

To start the external proxy:

- Execute the `cluster-proxy` script in the `bin` directory of the GraphDB distribution,
- Provide the cluster secret,
- 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d</code>, <code>--daemon</code></td>
<td>Daemonize (run in background)</td>
</tr>
<tr>
<td><code>-r</code>, <code>--follower-retries &lt;num&gt;</code></td>
<td>Number of times to retry a request to a different node in the cluster</td>
</tr>
<tr>
<td><code>-g</code>, <code>--graphdb-hosts &lt;address&gt;</code></td>
<td>List of GraphDB nodes’ HTTP or RPC addresses that are part of the same cluster</td>
</tr>
<tr>
<td><code>-h</code>, <code>--help</code></td>
<td>Print command line options</td>
</tr>
<tr>
<td><code>-p</code>, <code>--pid-file &lt;file&gt;</code></td>
<td>Write PID to &lt;file&gt;</td>
</tr>
<tr>
<td><code>-Dprop</code></td>
<td>Set Java system property</td>
</tr>
<tr>
<td><code>-Xprop</code></td>
<td>Set non-standard Java system property</td>
</tr>
</tbody>
</table>

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
```

**Important:** 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 - leader changes, nodes being added or removed, nodes out of reach, etc. 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.
4.6.8.2 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.SSLEnabled=true -Dgraphdb.connector.scheme=https -Dgraphdb.connector.security=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.

4.6.9 Cluster security

4.6.9.1 Encryption

As there is a lot of traffic between the cluster nodes, it is important that it is encrypted. In order to do so, the following requirements need to be met:

- SSL/TLS should be enabled on all cluster nodes.
- The nodes’ certificates should be 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 the node to be used as a cluster one.

See how to set up certificate trust between the nodes here.

4.6.9.2 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; or if you have configured OpenID on one of the nodes, it needs to be configured on the rest of them as well.

4.7 Using a Cluster

Once created, a cluster can be used almost like a single GraphDB configuration. However, all write operations need to 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 import some data in our cluster.

1. On the leader node, in our case `http://graphdb1.example.com:7280`, go to Setup → Repositories and create a repository (for Location, select Local).

2. We can see 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.

**SPARQL Query & Update**

```
select * where {
  } limit 100
```

4.7.1 Using the GraphDB client API for Java

The GraphDB client API for Java is an extension of RDF4J’s HTTPRepository that adds support for automatic leader discovery.

You can create an instance of `GraphDBHTTPRepository` like this:

```java
package com.ontotext.example;

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").build();
        // Use repository...
    }
}
```
.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 needs to 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 be tried always, so it is recommended to specify at least two of them in case one of them is down.

GraphDB 10 includes an additional mechanism that allows using any of the cluster nodes with any standard client, e.g., 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:

```xml
<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.

### 4.7.2 Using a cluster with external proxy

The cluster can also be used through an external proxy. To do this, instead of the providing the GraphDB HTTP address, you need to 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`.
4.8 Secure GraphDB

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.

### 4.8.1 Enable security

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.

### 4.8.2 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.
4.8.3 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.

In the example above, all users will be able to read and write in the repository called “my_repo”, and read the “remote_repo” repository. They will also be able to create or delete connectors and toggle plugins for the “my_repo” repository.

Application settings allow you to 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.

4.8.4 Users and Roles

4.8.4.1 Create new user

This is the user creation screen.
Any user can have three different roles:

- **User** - can save SPARQL queries, graph visualizations or user-specific server side settings. Can also be given specific repository permissions.

- **Repository manager** - in addition to what a standard user can do, also has full read and write permission to all repositories. Can create, edit, and delete them. Can also access monitoring and configure whether the service reports anonymous usage statistics.

- **Admin** - can perform any server operation.

Regular users can be granted specific repository permissions. Granting a write permission to a user will mean that they can also read that repository.

If you want to allow a particular user global access to all repositories, you can do that by using the *Any data repository* checkbox.

### 4.8.4.2 Set password

The edit icon under *Actions* next to each user in the list will take you to the following screen:

**Edit user: admin**
The only difference between the Edit user and Create new user screens is that in Edit user, you cannot change the username.

## 4.9 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
```

## 4.10 Application Settings

Application settings help you to configure the default behavior of the GraphDB Workbench.

The Workbench interface has some useful options that change only the way you query the database, not changing the rest of the GraphDB behavior:

### Settings

- **Expand results over owl:sameAs** - 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 value** - 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 - This includes or excludes predicates from owl:, rdf:, rdfs:, sesame:, dul:, prov:, fibo:, wd:.

• Count total results in SPARQL editor - 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.

• Ignore shared saved queries in SPARQL editor - In the SPARQL editor, saved queries can be shared, and you can choose not to see them.

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 through Setup  User and access Edit user.

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.

4.11 Backing up and Restoring a Repository

4.11.1 Back up a repository

Repository backups allow you to revert a GraphDB repository to a previous state. The database offers different approaches of copying the repository state:

• Export the repository to an RDF file - this operation can run in parallel to read and write, but takes longer to complete.
• Back up a repository using the JMX interface or cURL.
• Copy the repository image directory to a backup - this is a much faster option, but in non-cluster setups it requires shutdown of the database process.

Note:  We recommend all repository backups to be scheduled during periods of lower user activity.

4.11.1.1 Export repository to an RDF file

The repository export works without having to stop GraphDB. This operation usually takes longer than copying the low level file system, because all explicit RDF statements must be serialized and de-serialized over HTTP. Once the export operation starts, all following updates will not be included in the dump. To invoke the export repository operation, several interfaces are available:

Option 1: Export the repository with the GraphDB Workbench.

Export the database contents using the Workbench. To preserve the contexts (named graph) when exporting/importing the whole database, use a context-aware RDF file format, e.g., TriG.

1. Go to Explore  Graphs overview.
2. Choose the files you want to export.
3. Click Export graph as TriG.
Option 2: Export all statements with cURL.

The repository SPARQL endpoint supports dumping all explicit statements (replace the repositoryId with a valid repository name) with:

```
curl -X GET -H "Accept:application/x-trig" "http://localhost:7200/repositories/repositoryId/statements?\~\~infer=false" > export.trig
```

This method streams a snapshot of the database’s explicit statements into the `export.trig` file.

Option 3: Export all statements using the RDF4J API.

The same operation can be executed once with Java code by calling the `RepositoryConnection.exportStatements()` method with the `includeInferred` flag set to `false` (to return only the explicit statements).

Example:

```java
RepositoryConnection connection = repository.getConnection();
FileOutputStream outputStream = new FileOutputStream("export.nq");
RDFWriter writer = Rio.createWriter(RDFFormat.NQUADS, outputStream);
connection.exportStatements(null, null, null, false, writer);
IOUtils.closeQuietly(outputStream);
```

The returned iterator can be used to visit every explicit statement in the repository. One of the RDF4J RDF writer implementations can be used to output the statements in the chosen format.

**Note:** If the data will be re-imported, we recommend the N-Quads format as it can easily be broken down into large ‘chunks’ that can be inserted and committed separately.
4.11.1.2 Back up a repository using JMX interface

GraphDB offers backing up a repository through JMX.

Use the `OwlimRepositoryManager` MBean method `createZipBackup(String backupName)` with a `backupName` argument. This will create a zip file named `rep_<repository_id>-<timestamp>_backup.zip` with the content of the repository data directory (the storage folder is a subfolder there, so the `config.ttl` should be archived too). By default, it will be created in the `backup/backupName` directory of the repository’s folder.

You can also change the location of the backup directory by using the runtime property `-Dgraphdb.backup.base.folder=<full_path_to_target_folder>`. This will result in creating backup in the `<full_path_to_target_folder>/backup/backupName>` folder.

Invoking method with `null` parameter for `backupName` will result in creating backup in the `default` folder.

---

**Note:** Any attempt to create backup with an invalid `backupName` will result in the following message:

“Backup name must start with a letter, digit, or underscore.
Each subsequent character may be a letter, digit, underscore, dash, or period.”

---

You can invoke the method from JConsole, or by sending an HTTP request via cURL:

**Back up a repository from JConsole**

Invoke backup from the JMX interface using JConsole:
Back up a repository using cURL

```
curl -H 'content-type: application/json' -d "{"type":"exec","mbean":"com.ontotext:type=OwlimRepositoryManager,name="Repository (/full_path_to_repository_storage/)",operation":"createZipBackup","arguments":["backupName"]}" http://localhost:7200/jolokia/
```

Here is an example where `full_path_to_repository_storage` is replaced by a real path:

```
curl -H 'content-type: application/json' -d "{"type":"exec","mbean":"com.ontotext:type=OwlimRepositoryManager,name="Repository (/home/ubuntu/graphdb-se-8.7.0/data/repositories/test/storage/)",operation":"createZipBackup","arguments":["backupName"]}" http://localhost:7200/jolokia/
```

This will produce a folder `backupName` in the `<test>/backup/` directory which contains the backup zip.
4.11.1.3 Back up GraphDB by copying the binary image

Note: This is the fastest method to back up a repository, but it requires stopping the database.

All RDF data is stored only in your repository.

1. Stop the GraphDB server.
2. Manually copy the storage folders to the backup location.

```
kill <pid-of-graphdb>
sleep 10 #wait some time for the database to stop
cp -r {graphdb.home.data}/repositories/your-repo backup-dest/date/ #copies GraphDB's data
```

Tip: For more information about the data directory, see here.

4.11.2 Restore a repository

The restore options depend on the backup format.

Option 1: Restore a repository from an RDF export.

This option will import a previously exported file into an empty repository.

1. Make sure that the repository is empty or recreated with the same repository configuration settings.
2. Go to Import, and select the Server files tab.
3. Press the Help button to see the directory path where you need to import your files or directories.
4. Copy the RDF file with the backup into this directory path and refresh the Workbench.
5. Start the file import and wait for the data to be imported.

Option 2: Restore the database from a binary image or zip backup.

1. Make sure that the repository is empty or recreated with the same repository configuration settings.
2. Stop the GraphDB server.
3. Delete the entire your-repo folder, and copy/paste the folder of the data/repositories/your-repo from the backup copy.
4. Start the GraphDB server.
5. Run a quick test read query to make sure that the repository is initialized correctly.
4.12 Query Monitoring and Termination

Query monitoring and termination can be done manually from the Workbench or by running a JMX operation, and automatically by configuring GraphDB to abort queries after a certain query timeout is reached.

4.12.1 Query monitoring and termination using the Workbench

When there are running queries, their number is shown up next to the Repositories dropdown menu. To track and interrupt long running queries:

1. Go to Monitoring Queries or click the Running queries status next to the Repositories dropdown menu.
2. Press the Abort query button to stop a query.

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>the ID of the query</td>
</tr>
<tr>
<td>node</td>
<td>local or remote worker node repository ID</td>
</tr>
<tr>
<td>type</td>
<td>the operation type QUERY or UPDATE</td>
</tr>
<tr>
<td>query</td>
<td>the first 500 characters of the query string</td>
</tr>
<tr>
<td>lifetime</td>
<td>the time in seconds since the iterator was created</td>
</tr>
<tr>
<td>state</td>
<td>the low level details for the current query collected over the JMX interface</td>
</tr>
</tbody>
</table>

To interrupt long running queries, click the Abort query button.

You can also interrupt a query directly from the SPARQL Editor:
4.12.2 Query monitoring and termination using the JMX interface

4.12.2.1 Query monitoring

GraphDB offers a number of monitoring and control functions through JMX. It also provides detailed statistics about executing queries, or, more accurately, query result iterators. This is done through the RepositoryMonitor MBean, one for each repository instance. Each bean instance is named after the storage directory of the repository to which it relates.

<table>
<thead>
<tr>
<th>Package</th>
<th>com.ontotext</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBean name</td>
<td>RepositoryMonitor</td>
</tr>
</tbody>
</table>

The RepositoryMonitor MBean has two attributes - TrackRecords and TrackRecordsNumber.

The TrackRecords attribute is an array of objects with the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msSinceCreated</td>
<td>the time (in ms) since the iterator was created</td>
</tr>
<tr>
<td>numberOfOperations</td>
<td>the total number of operations for this iterator</td>
</tr>
<tr>
<td>requestedToStop</td>
<td>indicates if the query has been requested to terminate early (see below)</td>
</tr>
<tr>
<td>sparqlString</td>
<td>the full text of the query</td>
</tr>
<tr>
<td>state</td>
<td>the current state of the iterator: ACTIVE, IN_HAS_NEXT, COMMIT_PENDING, IN_COMMIT, IN_COMMIT_PLUGIN, IN_NEXT, BEGIN_PENDING, IN_PARALLEL_IMPORT, IN_PARALLEL_COMMIT, ENQUEUED, CLOSED</td>
</tr>
<tr>
<td>trackAlias</td>
<td>a unique alias for this iterator given by the client</td>
</tr>
<tr>
<td>trackId</td>
<td>a unique ID for this iterator - if debug level is used to increase the detail of the GraphDB output, then this value is used to identify queries when logging the query execution plan and optimization information</td>
</tr>
<tr>
<td>type</td>
<td>the type of this iterator - UPDATE or READ</td>
</tr>
</tbody>
</table>

The TrackRecordsNumber attribute is the number of running queries.
The collection of these objects grows for each executing/executed query. However, older objects in the CLOSED state expire and are removed from the collection, as the query result iterators are garbage collected.

### 4.12.2.2 Terminating a query

The following operations are available with `RepositoryMonitor` MBean:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>requestStop</code></td>
<td>Requests that a query terminates early; parameter: trackId of the query to stop</td>
</tr>
<tr>
<td><code>requestStopByAlias</code></td>
<td>Requests that a query terminates early; parameter: trackAlias of the query to stop</td>
</tr>
<tr>
<td><code>getFullSparqlString</code></td>
<td>Returns the full text of the query; parameter: trackId of the query</td>
</tr>
</tbody>
</table>

The `requestStop` and `requestStopByAlias` operations allow an administrator to request earliest as possible termination of a query.

To terminate a query, execute the `requestStop` command with given `trackId` of the query; or `requestStopByAlias` with given `trackAlias`.

As a result:

- The `requestedToStop` attribute is set to `true`.
- The query terminates normally when `hasNext()` returns `false`.
- The so far collected result will be returned by the interrupted query.
4.12.3 Terminating a transaction

It is also possible to terminate a long committing update transaction. For example, when committing a ‘chain’ of
many thousands of statements using some transitive property, the inferencer will attempt to materialize all possible
combinations leading to hundreds of millions of inferred statements. In such a situation, you can abort the commit
operation and roll back to the state the database had before the commit was attempted.

The following MBean is used:

<table>
<thead>
<tr>
<th>Package</th>
<th>com.ontotext</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBean name</td>
<td>OwlimRepositoryManager</td>
</tr>
</tbody>
</table>

This MBean has no attributes:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abortTransaction</td>
<td>Requests that the currently executing (lengthy) commit operation be terminated and rolled back.</td>
</tr>
</tbody>
</table>

4.12.4 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 the number of seconds you have set in it, where a default value of 0 indicates no limit.

4.13 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:

- Configure GraphDB memory
- Data loading & query optimizations
  - Dataset loading
  - GraphDB’s optional indexes
  - Cache/index monitoring and optimizations
  - Query optimizations
- Explain Plan
- Inference optimizations
  - Delete optimizations
  - Rules optimizations
  - Optimization of owl:sameAs
  - RDFS and OWL support optimizations
4.13.1 Data loading & query 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.

4.13.1.1 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:

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>enablePredicateList = false</td>
</tr>
<tr>
<td>enable-context-index = false</td>
</tr>
<tr>
<td>in-memory-literal-properties</td>
</tr>
</tbody>
</table>

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:
### Table of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>entity-index-size</strong></td>
<td>Set to a large enough value.</td>
</tr>
<tr>
<td><em>(see more)</em></td>
<td></td>
</tr>
<tr>
<td><strong>enablePredicateList</strong></td>
<td>Can speed up queries (and loading).</td>
</tr>
<tr>
<td><em>(see more)</em></td>
<td></td>
</tr>
<tr>
<td><strong>enable-context-index</strong></td>
<td>Provides better performance when executing queries that use contexts.</td>
</tr>
<tr>
<td><em>(see more)</em></td>
<td></td>
</tr>
<tr>
<td><strong>index-in-memory-literal-properties</strong></td>
<td>Defines whether to keep the properties of each literal in-memory.</td>
</tr>
<tr>
<td><em>(see more)</em></td>
<td></td>
</tr>
</tbody>
</table>

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.

#### 4.13.1.2 GraphDB's optional 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.

---

4.13. Performance 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:

<table>
<thead>
<tr>
<th>Package</th>
<th>com.ontotext</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBean name</td>
<td>CollectionStatistics</td>
</tr>
</tbody>
</table>

The following information is displayed for each MBean/index:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheHits</td>
<td>The number of operations completed without accessing the storage system.</td>
</tr>
<tr>
<td>CacheMisses</td>
<td>The number of operations completed, which needed to access the storage system.</td>
</tr>
<tr>
<td>FlushInvocations</td>
<td></td>
</tr>
<tr>
<td>FlushReadItems</td>
<td></td>
</tr>
<tr>
<td>FlushReadTimeTotal</td>
<td></td>
</tr>
<tr>
<td>FlushWriteItems</td>
<td></td>
</tr>
<tr>
<td>FlushWriteTimeAverage</td>
<td></td>
</tr>
<tr>
<td>FlushWriteTimeTotal</td>
<td></td>
</tr>
<tr>
<td>PageDiscards</td>
<td>The number of times a non-dirty page’s memory was reused to read in another page.</td>
</tr>
<tr>
<td>PageSwaps</td>
<td>The number of times a page was written to the disk, so its memory could be used to load another page.</td>
</tr>
<tr>
<td>Reads</td>
<td>The total number of times an index was searched for a statement or a range of statements.</td>
</tr>
<tr>
<td>Writes</td>
<td>The total number of times a statement was added to a collection.</td>
</tr>
</tbody>
</table>

The following operations are available:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetCounters</td>
<td>Resets all the counters for this index.</td>
</tr>
</tbody>
</table>

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.

### 4.13.1.4 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 GraphDB’s `Explain Plan` for a way to view query plans and applied optimizations.

#### 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, e.g., using the following SPARQL query construct:

```
FILTER ( langMatches(lang(?name), "es") )
```

In this situation, the `in-memory-literal-properties` configuration parameters can be set to `true`, causing the data values with language tags to be cached.

#### 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 (see Other special query behavior) to significantly reduce these duplicate results (by returning a single representative from each equivalence class).

Consider these example queries executed against the FactForge combined dataset. Here, the default is to enumerate:

```sparql
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * WHERE { ?c rdfs:subClassOf dbpedia:Airport}
```

producing many results:

```
dbpedia:Air_strip
http://sw.cyc.com/concept/Mx4ruQSTAL_QQdeZXF-MIWWdng
umbel-sc:CommercialAirport
opencyc:Mx4ruQSTAL_QQdeZXF-MIWWdng
dbpedia:Jetport
dbpedia:Airstrips
dbpedia:Airport
fb:guid.9202a8c04000641f80000000004ae12
opencyc-en:CommercialAirport
```

If you specify the `onto:disable-sameAs` pseudo-graph:

```sparql
PREFIX onto: <http://www.ontotext.com/> 
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT * FROM onto:disable-sameAs
WHERE { ?c rdfs:subClassOf dbpedia:Airport}
```

only two results are returned:
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, e.g., `filter(strstarts(str(?x), "http://dbpedia.org/ontology"))`, this may omit some valid solutions, as not all URIs within an equivalence class are matched against the filter.

### 4.13.1.5 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:

```sparql
INSERT DATA {
}
```

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.

### 4.13.2 Explain Plan

#### 4.13.2.1 What is GraphDB’s 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. It can help you improve your query, leading to better execution performance.

#### 4.13.2.2 Activating the explain plan

To see the query explain plan, use the `onto:explain` pseudo-graph:

```sparql
PREFIX onto: <http://www.ontotext.com/>
SELECT * FROM onto:explain
```
4.13.2.3 Simple explain plan

For the simplest query explain plan possible (\(?s \ ?p \ ?o\)), execute the following query:

```sql
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:

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. See below the triple patterns in the order in which they are executed internally.

- ----- Begin optimization group 1 -----: indicates starting 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.
4.13.2.4 Multiple triple patterns

**Note:** The result of the explain plan is given in the exact order, in which the engine will execute the query.

The following is an example where the engine reorders the triple patterns based on their complexity. The query is a simple join:

```sql
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 rdfs:subPropertyOf ?o2
}
```

and the output is:

```
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: 0 Current complexity: 10  
  # ----- 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.
```

Understanding 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.
4.13.2.5 Wine queries

All of the following examples are based on this simple dataset describing five fictitious wines. The file is quite small and contains the following data:

- 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.

Query 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.

```sparql
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.ontotext.com/example/wine#>
SELECT (COUNT(?wine) as ?wines) ?year
FROM onto:explain
WHERE {
  OPTIONAL {
    ?wine wine:hasYear ?year
  }
}
GROUP BY ?year
ORDER BY DESC(?wines)
```

When you execute the query in GraphDB, you get the following as an output (instead of the real results):

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
1  PREFIX wine: <http://www.ontotext.com/example/wine#>
1  SELECT (COUNT(?wine) AS ?wines) ?year
1  {
1    { # Begin optimization group 1 -------
1    } # End optimization group 1 -------
1    # ESTIMATED NUMBER OF ITERATIONS: 5
1    OPTIONAL {
1      { # Begin optimization group 2 -------
1        ?wine wine:hasYear ?year . # Collection size: 5 Predicate collection size: 5 Unique subjects: 5 Unique objects: 2 Current complexity: 1
1      } # End optimization group 2 -------
1      # ESTIMATED NUMBER OF ITERATIONS: 1
1    }
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.
Query with filter aggregation

This aggregation query applies a filter to the result set after grouping via the `HAVING` clause. It retrieves red wines made from more than one type of grape along with their grapes count.

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.ontotext.com/example/wine#>
PREFIX onto: <http://www.ontotext.com/>
SELECT ?wine (COUNT(?grape) AS ?grapeCount)
FROM onto:explain
WHERE {
    ?wine rdf:type wine:RedWine ;
    wine:isMadeFromGrape ?grape .
} GROUP BY ?wine
HAVING (?grapeCount > 1)
```

The returned explain plan will be:

```
plan 0

{ PREF fix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.ontotext.com/example/wine#>
SELECT ?wine (COUNT(?grape) AS ?grapeCount)
{
    # Begin optimization group 1
    ?wine rdf:type wine:RedWine ;
    wine:isMadeFromGrape ?grape .
} # End optimization group 1
FILTER (?grapeCount > 1)
GROUP BY ?wine

# 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.
```

Query with filter function

This is a typical SPARQL query with filter function. It retrieves the wines that are made from Pinot Noir grape.

```sparql
PREFIX onto: <http://www.ontotext.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX wine: <http://www.ontotext.com/example/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:isMadeFromGrape ?grape .
    ?grape rdfs:label ?grapeLabel .
    FILTER (?grapeLabel = "Pinot Noir")
}
```

And the output will be:
4.13.3 Inference optimizations

4.13.3.1 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_18:**

```
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 proton: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.

4.13.3.2 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: ...
```

(continues on next page)
Time overall (all rules): ... ns.

• A tabular format of the descriptive rule stats is also supported, e.g.:

<table>
<thead>
<tr>
<th>Repository name: university</th>
</tr>
</thead>
<tbody>
<tr>
<td>RULE;TIME;INVOKED;ITERNEXTS;FIRED;FIRETIME;INFERRED</td>
</tr>
<tr>
<td>prp_spo1_1; 36973700; 66; 81; 81; 36917600; 28</td>
</tr>
<tr>
<td>prp_spo1_0; 34354600; 17; 83; 83; 34310500; 25</td>
</tr>
</tbody>
</table>

• 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 <top:premise> p
 t <top:restriction> r
 t <top:conclusion> q
 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_premise b 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 1,981,987,589 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:

<table>
<thead>
<tr>
<th>rule</th>
<th>ver</th>
<th>tried</th>
<th>time</th>
<th>patts</th>
<th>checks</th>
<th>time</th>
<th>fired</th>
<th>time</th>
<th>triples</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptop_PropChain</td>
<td>4</td>
<td>163475763</td>
<td>776.3</td>
<td>5</td>
<td>117177482</td>
<td>185.3</td>
<td>15547176</td>
<td>590.9</td>
<td>9707142</td>
<td>12505</td>
</tr>
</tbody>
</table>

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule</td>
<td>the rule ID (name)</td>
</tr>
<tr>
<td>ver</td>
<td>the rule version (variant) or “T” for overall rule totals</td>
</tr>
<tr>
<td>tried, time</td>
<td>the number of times the rule/variant was tried, the overall time in sec</td>
</tr>
<tr>
<td>patts, time</td>
<td>the number of triple patterns (premises) in the rule, not counting the leading premise</td>
</tr>
<tr>
<td>checks, time</td>
<td>the number of times premises were checked, time in sec</td>
</tr>
<tr>
<td>fired</td>
<td>the number of times all premises matched, so the rule was fired</td>
</tr>
<tr>
<td>triples</td>
<td>the number of inferred triples</td>
</tr>
<tr>
<td>speed</td>
<td>inference speed, triples/sec</td>
</tr>
</tbody>
</table>

Run the script the following way:

```
```
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.

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.
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
```
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:premise>  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:premise>  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 rule sets 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 special GraphDB query behavior`). 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:

```
dcterm:created rdfs:subPropertyOf dc:date, dcterms:date.
dcterms:date rdfs:subPropertyOf dc:date.
```

This means that every `dcterm: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:

- 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`).

4.13. Performance Optimizations 129
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 this draft for some ideas and clear illustrations. Below, we describe three of these ideas.

**Tip:** To learn more, see a detailed account of applying some of these ideas in a real-world setting. Here are the respective rule implementations.

### 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:

<table>
<thead>
<tr>
<th>Id: ptop_PropChain</th>
</tr>
</thead>
<tbody>
<tr>
<td>t <a href="">ptop:premise1</a> p1</td>
</tr>
<tr>
<td>t <a href="">ptop:premise2</a> p2</td>
</tr>
<tr>
<td>t <a href="">ptop:conclusion</a> q</td>
</tr>
<tr>
<td>t <a href="">rdf:type</a> <a href="">ptop:PropChain</a></td>
</tr>
<tr>
<td>x p1 y</td>
</tr>
<tr>
<td>y p2 z</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>x q z</td>
</tr>
</tbody>
</table>

It is used with axiomatic triples as in the following:

```sparql
@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:

<table>
<thead>
<tr>
<th>Id: psys_transitiveOver</th>
</tr>
</thead>
<tbody>
<tr>
<td>p <a href="">psys:transitiveOver</a> q</td>
</tr>
<tr>
<td>x p y</td>
</tr>
<tr>
<td>y q z</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>x p z</td>
</tr>
</tbody>
</table>

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:
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.:

```
```

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 rule sets 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:

4.13. Performance Optimizations
• 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’ rule sets, 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.
4.13.3.3 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
- Symmetry, 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 systemgraph;`
- (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:

```xml
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:

```xml
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:

```xml
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 [here].
How `disable-sameAs` interferes with the different rulesets

The following parameters can affect the `owl:sameAs` behavior:

- **ruleset** – `owl:sameAs` support is enabled for all rulesets, except 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:e 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:d .
test:a owl:sameAs test:e .
test:b a rdf:Property .
test:b rdfs:subPropertyOf test:b .
```

(continues on next page)
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:e 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 onto: <http://www.ontotext.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
  }
}
```

If you have started with owl-horst-optimized, disable-sameAs=false, you will receive:
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:

```sparql
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:

```sparql
: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>.

4.13.3.4 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 `X rdfs: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 rule sets are provided. These have `-optimized` appended to the ruleset name, e.g., `owl-horst-optimized`.

The following optimizations are enacted in GraphDB:

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Affects patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove axiomatic triples</td>
<td>- &lt;any&gt; &lt;any&gt; <a href="">rdfs:Resource</a></td>
</tr>
<tr>
<td></td>
<td>- <a href="">rdfs:Resource</a> &lt;any&gt; &lt;any&gt;</td>
</tr>
<tr>
<td></td>
<td>- &lt;any&gt; <a href="">rdfs:domain</a> <a href="">rdf:Property</a></td>
</tr>
<tr>
<td></td>
<td>- &lt;any&gt; <a href="">rdfs:range</a> <a href="">rdf:Property</a></td>
</tr>
<tr>
<td></td>
<td>- <a href="">owl:sameAs</a> <a href="">rdf:type</a> <a href="">owl:SymmetricProperty</a></td>
</tr>
<tr>
<td></td>
<td>- <a href="">owl:sameAs</a> <a href="">rdf:type</a> <a href="">owl:TransitiveProperty</a></td>
</tr>
<tr>
<td>Remove rule conclusions</td>
<td>- &lt;any&gt; &lt;any&gt; <a href="">rdfs:Resource</a></td>
</tr>
<tr>
<td>Remove rule constraints</td>
<td>- [Constraint &lt;variable&gt; != <a href="">rdfs:Resource</a>]</td>
</tr>
</tbody>
</table>

4.14 Troubleshooting

4.14.1 Database 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
4.14.1.1 Possible values for health checks and their meaning

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository-state</td>
<td>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.</td>
</tr>
<tr>
<td>read-availability</td>
<td>Checks whether the repository is readable.</td>
</tr>
<tr>
<td>storage-folder</td>
<td>Checks if there are at least 20 megabytes writable left for the storage folder. The amount can be controlled with the system parameter health.minimal.free.storage.</td>
</tr>
<tr>
<td>long-running-queries</td>
<td>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.</td>
</tr>
<tr>
<td>predicates-statistics</td>
<td>Checks if the predicate statistics contain correct values.</td>
</tr>
<tr>
<td>plugins</td>
<td>Provides aggregated health checks for the individual plugins.</td>
</tr>
</tbody>
</table>

4.14.1.2 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:

```json
{
    "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": "plugins",
            "status": "yellow",
            "components": [
                {
                    "name": "elasticsearch-connector",
                    "status": "green",
                    "components": [
                    ]
                }
            ]
        }
    ]
}
```

(continues on next page)
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)
- **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:

```bash
curl 'http://localhost:7200/repositories/myrepo/health?'
```

4.14.1.3 Running 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):

```json
{
  "status": "green",
  "components": [
    {
      "status": "green",
      "name": "repository-state",
      "message": "INACTIVE"
    }
  ],
  "name": "test"
}
```

Example health from a passive check (repo is currently initializing):

```json
{
  "status": "yellow",
  "components": [
    {
      "status": "yellow",
      "name": "repository-state",
      "message": "STARTING"
    }
  ],
  "name": "test"
}
```

Note: From GraphDB 9.7.x onwards, legacy health checks are no longer supported.
4.14.2 System metrics 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>`.

### 4.14.2.1 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache.flush</td>
<td>A timer 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.</td>
</tr>
<tr>
<td>cache.hit</td>
<td>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.</td>
</tr>
<tr>
<td>cache.load</td>
<td>A timer for the pages that have to be read from the disc. The smaller the number of pages is, the better.</td>
</tr>
<tr>
<td>cache.miss</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 4.14.2.2 Entity pool metrics

You can monitor the number of reads and writes in the entity pool of each repository with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epool.read</td>
<td>A timer for the number of reads in the entity pool.</td>
</tr>
<tr>
<td>epool.write</td>
<td>A timer for the number of writes in the entity pool.</td>
</tr>
</tbody>
</table>
4.14.3 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.

### 4.14.3.1 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.

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 with a default of 60 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`.

**4.14.3.2 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 we 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
  ```

**4.14. Troubleshooting**
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.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>audit-log.log</td>
<td>On a running GraphDB instance with security ON, this file contains a log of all operations performed on the instance and who performed them. To start an audit trail, at least one of the <code>graphdb.audit.role</code> and <code>graphdb.audit.repository</code> parameters should be enabled.</td>
</tr>
<tr>
<td>error.log</td>
<td>Contains a log of all ERROR messages returned while the instance was running.</td>
</tr>
<tr>
<td>main.log</td>
<td>Contains all messages coming from the main part of the engine.</td>
</tr>
<tr>
<td>query-log.log</td>
<td>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.</td>
</tr>
<tr>
<td>slow-query-log.log</td>
<td>Contains slow queries as per the <code>SlowOpThresholdMs</code> parameter.</td>
</tr>
</tbody>
</table>

4.14.4 Storage tool

The Storage Tool is an application for scanning and repairing a GraphDB repository. To run it, execute `bin/storage-tool` 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).
4.14.4.1 Options

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>Operation to be executed, mandatory</td>
<td></td>
</tr>
<tr>
<td>storage</td>
<td>Absolute path to repo storage directory, mandatory</td>
<td></td>
</tr>
<tr>
<td>esize</td>
<td>Size of entity pool IDs: 32 or 40 bits</td>
<td>32</td>
</tr>
<tr>
<td>statusPrintInterval</td>
<td>Interval between status message printing</td>
<td>38, means 30 seconds</td>
</tr>
<tr>
<td>pageCacheSize</td>
<td>Size of the page cache</td>
<td>10, means 10,000 elements</td>
</tr>
<tr>
<td>sortBufferSize</td>
<td>Size of the external sort buffer</td>
<td>95, means 95M elements, max value is also 95</td>
</tr>
<tr>
<td>positiveFilterStatus</td>
<td>Optional statement status filter during export</td>
<td>-1, means no filter</td>
</tr>
<tr>
<td>srcIndex</td>
<td>One of pso, pos</td>
<td></td>
</tr>
<tr>
<td>destIndex</td>
<td>One of pso, pos, cpso</td>
<td></td>
</tr>
<tr>
<td>origURI</td>
<td>Original existing URI in the repo</td>
<td></td>
</tr>
<tr>
<td>replURI</td>
<td>New non-existing URI in the repo</td>
<td></td>
</tr>
<tr>
<td>destFile</td>
<td>Path to file used to store exported data</td>
<td></td>
</tr>
</tbody>
</table>

4.14.4.2 Supported commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>Scans repo index(es) and prints statistic of the number of statements and repo consistency.</td>
</tr>
<tr>
<td>rebuild</td>
<td>Uses the source index (srcIndex) to rebuild the destination index destIndex. If srcIndex = destIndex, compacts destIndex. If srcIndex is missing and destIndex = predicates, then it just rebuilds destIndex.</td>
</tr>
<tr>
<td>replace</td>
<td>Replaces an existing entity -origURI with a non-existing one -replURI.</td>
</tr>
<tr>
<td>repair</td>
<td>Repairs the repository indexes and restores data, a better variant of the merge index.</td>
</tr>
<tr>
<td>mergeindex</td>
<td>Merges pso and pos indexes (makes union), rebuilds context indexes if any. Note that there is no data backup.</td>
</tr>
<tr>
<td>export</td>
<td>Uses the source index (srcIndex) to export repository data to the destination file (destFile). Supported destination file extension formats: .trig, .ttl, .nq.</td>
</tr>
<tr>
<td>epool</td>
<td>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.</td>
</tr>
</tbody>
</table>

4.14.4.3 Examples

- scan the repository, print statement statistics and repository consistency status:

```
bin/storage-tool -command=scan -storage=/repo/storage
```

when everything is OK

```
Scan result consistency check!

mask | pso | pos | diff | flags
-----|-----|-----|------|-----
0001 | 29,937,266 | 29,937,266 | OK   | INF |
0002 | 61,251,858 | 61,251,858 | OK   | EXP |
```

(continues on next page)
Scan determines that this repo image is consistent.

- when there are broken indexes

Scan determines that this repo image is INCONSISTENT.

Literals index contains more statements than the literals in epool, and you have to rebuild it:

- scan the PSO index of a 40bit repository, print a status message every 60 seconds:
4.14. Troubleshooting

- compact the PSO index (self-rebuild equals compacting):
  ```
  bin/storage-tool -command=rebuild -storage=/repo/storage -esize=40 -srcIndex=pso -destIndex=pso
  ```

- rebuild the POS index from the PSO index and compact POS:
  ```
  bin/storage-tool -command=rebuild -storage=/repo/storage -esize=40 -srcIndex=pso -destIndex=pos
  ```

- rebuild the predicates statistics index:
  ```
  bin/storage-tool -command=rebuild -storage=/repo/storage -esize=40 -destIndex=predicates
  ```

- replace http://onto.com#e1 with http://onto.com#e2:
  ```
  bin/storage-tool -command=replace -storage=/repo/storage -origURI="<http://onto.com#e1>" -replURI="<http://onto.com#e2>"
  ```

- dump the repository data using the POS index into a f.trig file:
  ```
  bin/storage-tool -command=export -storage=/repo/storage -srcIndex=pos -destFile=/repo/storage/f.trig
  ```

- scan the entity pool and create report with invalid IRIs, if such exist:
  ```
  bin/storage-tool -command=epool -storage=/repo/storage -esize=40
  ```
5.1 Loading Data

GraphDB exposes multiple interfaces for loading RDF data.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Use cases</th>
<th>Mode</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARQL endpoint</td>
<td>No limits on the file size</td>
<td>Online parallel</td>
<td>Moderate speed</td>
</tr>
<tr>
<td>Workbench import a local or a remote RDF file</td>
<td>Small files limited up to 200MB</td>
<td>Online parallel</td>
<td>Moderate speed</td>
</tr>
<tr>
<td>Workbench import a server file</td>
<td>No limits on the file size</td>
<td>Online parallel</td>
<td>Fast ignoring all HTTP protocol overheads</td>
</tr>
<tr>
<td>ImportRDF Load</td>
<td>Batch import of very big files</td>
<td>Initial offline import with no plugins</td>
<td>Fast, with small speed degradation</td>
</tr>
<tr>
<td>ImportRDF Preload</td>
<td>Import huge datasets with no inference</td>
<td>Initial offline import with no inference and plugins</td>
<td>Ultra-fast without speed degradation</td>
</tr>
</tbody>
</table>

5.1.1 Loading data using the Workbench

There are several ways of importing 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);
- by pasting the RDF data in the Text area tab;
- from a SPARQL construct query directly.

All import methods support asynchronous running of the import tasks, except for the text area import, which is intended for very fast and simple import.

**Note:** Currently, only one import task of a type is executed at a time, while the others wait in the queue as pending.

**Note:** For local repositories, we support interruption and additional settings, since the parsing is done by the Workbench. When the location is a remote one, you just send the data to the remote endpoint, and the parsing and loading are performed there.

If you have many files, a file name filter is available to narrow the list down.
5.1.1.1 Import settings

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

- **Base IRI** - specifies 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.

- **Target graphs** - when specified, imports the data into one or more graphs. Some RDF formats may specify graphs, while others do not support that. The latter are treated as if they specify the default graph.
  - From data - Imports data into the graph(s) specified by the data source.
  - The default graph - Imports all data into the default graph.
  - Named graph - Imports everything into a user-specified named graph.

- **Enable replacement of existing data** - Enable this to replace the data in one or more graphs with the imported data. When enabled:
  - **Replaced graph(s)** - All specified graphs will be cleared before the import is run. If a graph 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.

**Advanced settings:**

- **Preserve BNnode IDs**: assigns its own internal blank node identifiers or uses the blank node IDs it finds in the file.
- **Fail parsing if datatypes are not recognized**: determines whether to fail parsing if datatypes are unknown.
- **Verify recognized datatypes**: verifies that the values of the datatype properties in the file are valid.
- **Normalize recognized datatypes values**: indicates whether recognized datatypes need to have their values be normalized.
- **Fail parsing if languages are not recognized**: determines whether to fail parsing if languages are unknown.
- **Verify language based on a given set of definitions for valid languages**: determines whether languages tags are to be verified.
- **Normalize recognized language tags**: indicates whether languages need to be normalized, and to which format they should be normalized.
- **Should stop on error**: determines whether to ignore non-fatal errors.
- **Force serial pipeline**: enforces the use of the serial pipeline when importing data.

**Note:** *Import without changing settings* will import selected files or folders using their saved settings or default ones.
5.1.1.2 Importing local files

Go to Import User data Upload RDF files.

This option allows you to select, configure, and import data from various formats.

**Note:** The limitation of this method is that it supports files of a limited size. The default is 200 megabytes, and is controlled by the `graphdb.workbench.maxUploadSize` property. The value is in bytes (-Dgraphdb.workbench.maxUploadSize=20971520).

Loading data from your local machine directly streams the file to the RDF4J's 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 from the header.
3. The import settings modal appears, just in case you want to add additional settings.
5.1.1.3 Importing remote content

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, it is used to detect the correct data type (e.g., http://linkedlifedata.com/resource/umls-concept/C0024117.rdf). Otherwise, you have to provide the Data Format parameter, which is sent as Accept header to the endpoint and then to the import loader.

5.1.1.4 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 very simple text import sends the data to the Repository Statements Endpoint.

5.1.1.5 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/.

If you want to tweak the directory location, see the graphdb.workbench.importDirectory system property. The directory is scanned recursively and all files with a semantic MIME type are visible in the Server files tab.
5.1.1.6 Import data with an INSERT query

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

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA {
  GRAPE <http://example> {
    dc:title "A new book";
    dc:creator "A.N.Other" .
  }
}
```

5.1.2 Loading data using the ImportRDF tool

ImportRDF is a tool designed for offline loading of datasets. It cannot be used against a running server. Rationale for an offline tool is to achieve an 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. It loads data in a new repository created from the Workbench or the standard configuration Turtle file found in `configs/templates`, or uses an existing repository. In the latter case, the repository data is automatically overwritten.

5.1.2.1 Load vs Preload

The ImportRDF tool supports two sub-commands - Load and Preload. In GraphDB 9.x and older, they correspond to the **LoadRDF** and the **Preload** tools, respectively.

Despite the many similarities between Load and Preload, such as the fact that both commands do parallel offline transformation of RDF files into GraphDB image, there are also substantial differences in their implementation. Load uses an algorithm very similar to online data loading. As the data variety grows, the loading speed starts to drop, 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.

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. Thus, 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 valid RDF, `.zip` and `.gz` files, and directories.
5.1.2.2 Load command line options

Usage: importrdf load [OPTION]... [FILE]...
FILE... Files to be loaded.
-c, --config-file <file> Repository defining .ttl file.
-f, --force Whether to overwrite the existing repository.
-h, --help Display this message and exit!
-i, --repository <repository-name> Name of an existing repository.
-m, --mode <serial|parallel> Singlethreaded (serial) or multithreaded (parallel) mode for parse/load/infer.
-p, --partial-load Whether to allow partial load of file that contains corrupt line.
-s, --stop-on-error Whether to stop process if the dataset contains a corrupt file.
-v, --verbose 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.

Note: For loading datasets larger than several billion RDF statements, consider using the Preload sub-command.

5.1.2.3 Preload command line options

Usage: importrdf preload [OPTION]... [FILE]...
-a, --iterator-cache <arg> Chunk iterator cache size. The value will be multiplied by 1024, default is 'auto', e.g., calculated by the tool.
-b, --chunk <arg> 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.
-c, --config-file <file> Repository defining .ttl file.
-f, --force Whether to overwrite the existing repository.
-h, --help Display this message and exit.
-i, --id <repository-id> Existing repository ID.
-q, --queue-folder <folder> Folder used to store temporary data.
-r, --recursive Whether to walk folders recursively.
-t, --parsing-tasks <num> Number of RDF parsers.
-x, --restart Whether to restart the load, ignoring any existing recovery points.
-y, --recovery-point-interval <sec> The interval at which recovery points are created.

Note: To be able to load data using the below methods, make sure to set up a valid GraphDB license after installing GraphDB.

There are two common cases for loading data with the ImportRDF tool:
Load data in a repository created from the Workbench

1. Configure ImportRDF repositories location by setting the property `graphdb.home.data` in `<conf/graphdb.properties`. If no property is set, the default repositories location will be the `data` directory of the GraphDB distribution.

2. Start GraphDB.

3. In a browser, go to the Workbench Web application using a URL of this form: `http://localhost:7200`. Substitute `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:

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

   or if using the preload sub-command:

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

8. Start GraphDB.

Load data in a new repository initialized by a config file

1. Stop GraphDB.

2. Configure ImportRDF repositories location by setting the property `graphdb.home.data` in `<conf/graphdb.properties`. If no property is set, the default repositories location will be the `data` directory of the GraphDB distribution.

3. Create a configuration file.

4. Start the bulk load with the following command:

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

   or if using the preload sub-command:

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

5. Start GraphDB.

5.1.2.4 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:

```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/trree/graphdb#>.
[ ] a rep:Repository ;

(continues on next page)
```
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" ;
  ]
].

5.1.2.5 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)>

The following options are used 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.

5.1.2.6 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 controls how many parallel threads parse the input files.

- **--queue-folder**: the parameter controls the file system location, where all temporary chunks are stored.
5.1.2.7 Resuming data loading with Preload

The loading of a huge dataset is a long batch processing, and every run may take many hours. Preload supports resuming of the process if something goes wrong (insufficient disk space, out of memory, etc.) and the loading is terminated abnormally. In this case, the data processing will restart from intermediate restore point instead of at 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 data resuming:

- `--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`.

5.2 Exploring data

5.2.1 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 has no ontology (hierarchy), the RDF classes are visualized as separate circles instead of nested ones.
5.2.1.1 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.

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.
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 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 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.
### 5.2.1.2 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, enabling you to further explore the class connectedness by clicking the green nodes (object property class).

- To administrate your graph view, use the toolbar options on the right-hand 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.

### 5.2.2 Class relationships

To explore the relationships between the classes, navigate to Explore Class relationships. You can see a complicated diagram, which by default is showing 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 be 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, you can see that Person is the class with the biggest number of links. It is very strongly connected to Feature and City, and most of the links are from Person. Also, you notice that all classes have many outgoing links to opengis:_Feature.
Left of the 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 on it to see the actual classes this class is linked to, again ordered by the number of links with the actual number shown. The direction of the links is also displayed.
Use the list of classes to control which classes to see in the diagram with the add/remove icons next to each class. Remove all classes with the X icon on the top right. The green background of a class indicates that the class is present in the diagram. You see that Person has much more connections to City than Village.
For each two classes in the diagram you can find the top predicates that connect them, again ordered and with the number of statements of this predicate and instances of these classes. **Person** is linked to **City** by the **birthPlace** and **deathPlace** predicates.
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 explore the class relationships of your data programmatically. To do so, go to the SPARQL tab of the Workbench menu and execute the following query:

```sparql
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 ;
}
```

(continues on next page)
5.2.3 Explore resources

5.2.3.1 Explore resources through the easy graph

Navigate to Explore Visual graph. Easy graph enables you to explore the graph of your data without using SPARQL. You see a search input field to choose a resource as a starting point for graph exploration.

A graph of the resource links is shown. Nodes that have the same type have the same color. All types for a node are listed when you hover over it. By default, what you see are the first 20 links to other resources ordered by RDF rank if present. See the settings below to modify this limit and the types and predicates to hide or see with preference.
The size of the nodes reflects the importance of the node 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. Another way to expand it is to double-click on it.
Click on the node to know more about a resource.
Sofia

Sofia (/ˈsɒfɪə/) (Bulgarian: София, Sofiya, pronounced [ˈsɔfijə]) is the capital and largest city of Bulgaria. Sofia is the 14th largest city in the European Union with a population of more than 1.2 million people. The city is located at the foot of Vitosha Mountain in the western part of the country, within less than 50 kilometres (31 mi) drive from the Serbian border.

Types:
- **dbo:City**
- schema:City
- gn:Feature
- geo-pos:SpatialThing
- wd:0515

RDF rank:

Search instance properties

- **dbo:PopulatedPlace/areaTotal**
  492.0

- **dbo:PopulatedPlace/populationDensity**
  2496.0

- **dbo:abstract**
  Sofia (/ˈsɒfɪə/) (Bulgarian: София, Sofiya, pronounced [ˈsɔfijə]) is the capital and largest city of Bulgaria. Sofia is the 14th largest city in the European Union with a population of more than 1.2 million people.
The side panel includes the following:

- a short description (rdfs:comment)
- labels (rdfs:label)
- RDF rank
- image (foaf:depiction) if present, and all DataType properties. You can search by DataType property if you are interested in a certain value.

You can click on the node again to hide the panel.

Note that 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.

Once a node is expanded, you have the option to collapse it. This will remove all its links and their nodes, except those that are connected to other nodes also. See the example below. Collapsing “Eastern European Time” removes all nodes except Bulgaria, because Bulgaria is also linked to Sofia which is expanded.
If you are not interested in a node anymore, you can hide it using the remove icon. The focus icon is used to restart the graph with the node of interest. Use carefully, since it resets the state of the graph.

More global actions are available in the menu in the upper right corner. Use the arrows to visually rotate your graph for convenience.

Click on the settings icon to configure your graph globally.
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 of importance in this case.
- **Show/hide predicate labels** is an option for convenience when you are not interested which predicates link the nodes.
- **Preferred** and **ignored** types/predicates is an advanced option. If you know your data well, you will be able to control to a bigger extent what to see when you expand nodes. If a preferred type is present, nodes of that type will be shown before all other types (see 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 [http://dbpedia.org/ontology/Person](http://dbpedia.org/ontology/Person) as preferred type and tick the option to see only preferred types. Only links to Person instances are shown, related to Sofia.
5.2.3.2 Create your own visual graph

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.

Create visual graph config

- **Config name**: My graph config (required)
- **Description**: Enter a description to guide users what this visual graph represents
- **Hint**: Add a hint to guide users what the starting point IRI should be

**Starting point**
- **Graph expansion**
- **Node basics**
- **Edge basics**
- **Node extra**

- Start with a search box: Choose the starting point of your visual graph every time
- Start with a fixed node: The visual graph will always start from the chosen node
- Start with graph query results: The results from your query will be the starting point of the visual graph

Share visual graph with other users

Save
The configuration consists of five queries separated in different tabs. A list of sample queries is provided to guide you in the process. Note that some bindings are required.

- **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. 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, i.e., 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 be replaced with 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 subject and object are nodes, and each 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 be replaced with 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 query SELECT 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. Explore your data with your custom visual graph.

5.2.3.3 Save and share graphs

During graph exploration, you can save a snapshot of the graph state with the Save icon in the top right to load it later. The graph config you are currently using is also saved, so when you load a saved graph you can continue exploring with the same config.

GraphDB also allows you to share your saved graphs with other users. When security is ON in Setup → Users and Access menu, the system distinguishes between different users. The graphs that you choose to share are only editable by you.

![Create new saved graph](image)

5.2. Exploring data 177
The graphs are located in Visual graph  Saved graphs. Other users will be able to view them and copy their URL by clicking the Get URL to graph icon.

When Users and Access  Free Access is ON, the free access user will see shared graphs only and will not be able to save new graphs.

5.2.4 View and edit resources

5.2.4.1 View and add a resource

**Important:** Before using the View resource functionality, 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, which 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.

**CabernetFrancGrape**

<table>
<thead>
<tr>
<th>subject</th>
<th>predicate</th>
<th>object</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>vin: CabernetFrancGrape</td>
<td>rdf:type</td>
<td>owl:Thing</td>
<td><a href="http://example.com/wine">http://example.com/wine</a></td>
</tr>
<tr>
<td>vin: CabernetFrancGrape</td>
<td>rdf:type</td>
<td>owl:Thing</td>
<td><a href="http://www.ontotext.com/explicit">http://www.ontotext.com/explicit</a></td>
</tr>
<tr>
<td>vin: CabernetFrancGrape</td>
<td>rdf:type</td>
<td>vin:WineGrape</td>
<td><a href="http://example.com/wine">http://example.com/wine</a></td>
</tr>
</tbody>
</table>

Even when the resource is not in the database, you can still add it from the resource view. Type in the resource IRI and hit Enter.
Here, you can create as many triples as you need for it, using the resource edit. 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.

**Edit MavrudGrape**

To view the new statements in TriG, click the **View TriG** button.

**Edit MavrudGrape**

When ready, save the new resource to the repository.
5.2.4.2 Edit a resource

Once you open a resource in View 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.

**Edit SweetRiesling**

Note: You cannot change or delete the inferred statements.

5.3 Querying Data

To manage and query your data, go to the SPARQL menu tab. The SPARQL view integrates the YASGUI query editor plus some additional features, which are described below.

Hint: 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.

The SPARQL editor offers two viewing/editing modes - horizontal and vertical.
SPARQL Query & Update

Use the vertical mode switch 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 switch between the different modes.

1. Manage your data by writing queries in the text area. It offers syntax highlighting and namespace autocompletion for easy reading and writing.

   **Tip:** To add/remove namespaces, go to Setup Namespaces.

2. 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).

3. 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.

4. Execute the query by clicking the Run button or use Ctrl/Cmd + Enter.
5. 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 order the results by column values and filter them by table values. The total number of results and the query execution time are displayed in the query results header.

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.

6. 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 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.

7. 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 (JSON, XML, CSV, TSV and Binary RDF for SELECT queries and all RDF formats for Graph query results).

5.3.1 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, and shared queries are editable by the owner only.

Access existing queries (default, yours, and shared) from the Show saved queries icon.

Copy your query as a URL by clicking the Get URL to current query icon.

When Free access is ON, the Free Access user will see shared queries only and will not be able to save new queries.
5.3.2 Interrupt queries

You can use the *Abort query* button in the SPARQL editor to manually interrupt any query.

![SPARQL Query & Update](image)

5.4 Exporting Data

Data can be exported in several ways and formats.

5.4.1 Exporting a repository

1. Go to *Explore  Graphs overview*.
2. Click *Export repository* and then the format that fits your needs.

![Explore](image)

5.4.2 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 above it.
3. Inspect a graph by clicking on it.
4. Delete a graph by clicking the bucket icon.
5. Or click to export the graph in the format of your choice.
5.4.3 Exporting query results

The SPARQL query results can also be exported from the SPARQL view by clicking Download As.

5.4.4 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:
5.5 Updating Data

5.5.1 Overview

Updating the content of RDF documents can generally be tricky due to the nature of RDF – no fixed schema or standard notion for management of multi-document graphs. There are two widely employed strategies when it comes to managing RDF documents – storing each RDF document in a single named graph vs. 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 needs to 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 (i.e., 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 – replace graph, SPARQL INSERT, add statements, etc.

5.5.1.1 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

5.5.1.2 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, e.g. timestamp information.
- The IRI of the updated document
- The new RDF contents of the updated document
5.5.2 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.

5.5.2.1 Direct access

Direct access is a direct connection to GraphDB using the RDF4J API as well as any GraphDB extensions to that API, e.g. using SPARQL, deleting/adding individual triples, etc.

Replace graph

When a replace graph smart update is sent directly to GraphDB, the user does not need to do anything special, e.g. 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:

```sparql
PREFIX factory: <http://factory/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
INSERT DATA {
    factory:1 rdf:type factory:Factory .
    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:

```sparql
SELECT * WHERE {
}
```

We will get the following result:

<table>
<thead>
<tr>
<th>p</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://factory/hasSalary">http://factory/hasSalary</a></td>
<td>&quot;10000&quot;^^<a href="http://www.w3.org/2001/XMLSchema#integer">http://www.w3.org/2001/XMLSchema#integer</a></td>
</tr>
<tr>
<td><a href="http://factory/worksIn">http://factory/worksIn</a></td>
<td><a href="http://factory/1">http://factory/1</a></td>
</tr>
</tbody>
</table>

4. Again in the SPARQL editor, create and execute the following template:

```sparql
INSERT DATA {
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX factory: <http://factory/>
    DELETE {
    }
    INSERT {
        ?id factory:updatedOn ?now
    }
}
```

(continues on next page)
5. Next, we 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 .
}
```

6. Now let’s see how the data has changed. Run again:

```
SELECT * WHERE {
}
```

We can see that John’s salary has increased.

<table>
<thead>
<tr>
<th>p</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://factory/hasSalary">http://factory/hasSalary</a></td>
<td>&quot;20000&quot;^^integer</td>
</tr>
<tr>
<td><a href="http://factory/worksIn">http://factory/worksIn</a></td>
<td><a href="http://factory/1">http://factory/1</a></td>
</tr>
</tbody>
</table>

### 5.5.2.2 Indirect access via Kafka Sink connector

In this mode, the user pushes update messages to Kafka and the Kafka Sink Connector 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 RDF formats.

For more details, see *Kafka Sink connector*.

Updates are performed as follows:

**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 recommended to use a unique ID, e.g. a random UUID.
- The Kafka value is the new RDF data to be added.

5.5.3 SPARQL templates

The built-in SPARQL template plugin enables you to create predefined SPARQL templates that can be used 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 the Workbench from both the SPARQL editor and from the SPARQL Templates editor.

5.5.3.1 From the SPARQL editor

Create template

We will use the template from the above example example. Execute:

```
INSERT DATA {
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX factory: <http://factory/>
    DELETE {
    } INSERT {
        ?id factory:updatedOn ?now
    } WHERE {
        ?id rdf:type factory:Factory .
        ?worker factory:worksIn ?id .
        bind(now() as ?now)
    }
    '"
}
```
Get template content

```sparql
SELECT ?template {
}
```

This will return the content of the template, in our case

```
*PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX factory: <http://factory/>
DELETE {
} INSERT {
    ?id factory:updatedOn ?now
} WHERE {
    ?id rdf:type factory:Factory .
    ?worker factory:worksIn ?id .
    bind(now() as ?now)
}
```

List defined templates

```sparql
SELECT ?id ?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:

```sparql
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 {
} INSERT {
    ?id factory:updatedOn ?now
} WHERE {
    ?id rdf:type factory:Factory .
    ?worker factory:worksIn ?id .
    bind(now() as ?now)
}
```

...```
Delete template

```
DELETE WHERE {
}
```

5.5.3.2 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 {
   } INSERT {
     ?id factory:updatedOn ?now
   } WHERE {
     ?id rdf:type factory:Factory .
     ?worker factory:worksIn ?id .
     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.

5.5.3.3 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 see how this is done.

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.ontotext.com/example/wine#> .
   ```
2. After that, let’s create the SPARQL template.

Go to Setup SPARQL Templates Create new SPARQL template and create the following template:

```
PREFIX wine: <http://www.ontotext.com/example/wine#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```
3. Let’s run a SPARQL query against the data. In the SPARQL editor, execute:

```sparql
PREFIX wine: <http://www.ontotext.com/example/wine#>
SELECT ?s ?p ?o
WHERE {
  BIND(wine:Blanquito as ?s).
}
```

The following results will be returned:

<table>
<thead>
<tr>
<th>s</th>
<th>p</th>
<th>t</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>onto:example/wine#Blanquito</td>
<td>rdf:type</td>
<td>onto:example/wine#</td>
<td>Blanquito</td>
</tr>
<tr>
<td>onto:example/wine#Blanquito</td>
<td>rdf:type</td>
<td>onto:example/wine#</td>
<td>WhiteWine</td>
</tr>
<tr>
<td>onto:example/wine#Blanquito</td>
<td>onto:example/wine#Grape</td>
<td>onto:example/wine#</td>
<td>Chardonnay</td>
</tr>
<tr>
<td>onto:example/wine#Blanquito</td>
<td>onto:example/wine#HasSugar</td>
<td>'dry'</td>
<td></td>
</tr>
<tr>
<td>onto:example/wine#Blanquito</td>
<td>onto:example/wine#HasYear</td>
<td>'2012'</td>
<td></td>
</tr>
</tbody>
</table>

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}/sparql­templates/execute.

b. For the repositoryID parameter, enter the name of your repository, e.g. “my_repo”.

c. In the document field, enter the JSON document:

```json
{
  "sugar": "none",
  "year": 2020,
  "s": "http://www.ontotext.com/example/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:

We can see that the objects for the sugar content and year predicates have been updated to “none” and “2020”, respectively.

Here, 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:

```json
{
  "sugar" : { "@id" : "custom:iri" },
  "year" : { "@value" : "2020",
             "@type" : "http://test.type" },
  "s" : "http://www.ontotext.com/example/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 returned results will be:
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. Here, this is demonstrated with the year variable - the result is "2020"^^<http://test.type>.

5.6 Kafka Sink Connector

**Note:** Despite having a similar name, the Kafka Sink connector is not a *GraphDB connector.*

### 5.6.1 Overview

Modern business has an ever increasing need of integrating data coming from multiple and diverse systems. Automating the update and continuous build of the knowledge graphs with the incoming streams of data can be cumbersome due to a number of reasons such as 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 and thus contributing to the automation of knowledge graph updates. It is a separately running process, which helps 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.

### 5.6.2 Setup

**Important:** Before setting up the connector, make sure to have JDK 11 installed.

1. Install Kafka 2.8.0 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.

### 5.6.3 Update types

The Kafka Sink connector supports three types of updates: simple add, replace graph, and 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 RDF formats.
5.6.3.1 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 that is to be added. 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, e.g. a random UUID.
- The Kafka value is the new RDF data to add.

Let’s see how it works.

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:

   ```bash
   sudo docker-compose up --scale graphdb=0
   ```

   where `graphdb=0` denotes that GraphDB must be started outside of the Docker container.
4. Next, we will configure the Kafka sink connector that will add data into the repository. In the directory of the Kafka sink connector, execute:

   ```bash
   curl http://localhost:8083/connectors \  
   --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",  
   "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"}}'
   ```

   with the following important parameters:

   - **topics**, which can be one of the following:
     - the name of the topic from which the connector will be reading the documents, here `gdb-add`
     - a comma-separated list of topics

   - **graphdb.server.url**: the URL of the GraphDB server, replace the sample value `http://graphdb.example.com:7200` with the actual URL

   **Important**: Since GraphDB is running outside the Kafka Sink Docker container, using localhost in `graphdb.server.url` will not work. Use a hostname or IP that is visible from within the container.

   - **graphdb.server.repository**: the GraphDB repository in which the connector will write the documents, here `kafka-test`
   - **graphdb.update.type**: the type of the update, here `ADD`
This will create the add data connector, which will read documents from the `gdb-add` topic and send them to the `test` repository on the respective GraphDB server.

**Note:** One connector can work with only one configuration. If multiple configurations are added, Kafka Sink will pick a single config and run it. If we need more than one connector, we have to create and configure them correspondingly.

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:

```
bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-add
```

6. To add some RDF data in the producer, paste this into the same window, and press Enter.

```
<http://example/subject> <http://example/predicate> "This is an example of adding data",
→ <http://example/graph>.
```

7. In the Workbench SPARQL editor of the “kafka-test” repository, run the query:

```
SELECT * WHERE {
  GRAPH ?g {
    ?s ?p ?o
  }
}
```

8. The RDF data that we just added via the producer should be returned as result.

### 5.6.3.2 Replace graph

In this update type, a document (the smallest update unit) is defined as the contents of a named graph. Thus, 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 as follows:

- 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 try it out.

1. We already have the Docker container with the Kafka sink connector running, and have created the “kafka-test” repository.
2. Now, let’s configure the Kafka sink connector that will replace data in a named graph. In the directory of the Kafka sink connector, execute:

```
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"},
```

(continues on next page)
with the same important parameters as in the add data example above.

This will configure the replace graph connector, which will read data from the \texttt{gdb-replace} topic and send them to the \texttt{kafka-test} repository on the respective GraphDB server.

\textbf{Note:} Here, we have created the connector on a different URL from the previous one - \url{http://localhost:8084/connectors}. If you want to create it on the same URL (\url{http://localhost:8083/connectors}), you need to first delete the existing connector:

\begin{verbatim}
    curl -X DELETE http://localhost:8083/connectors/kafka-sink-graphdb-add
\end{verbatim}

3. To replace data in a specific named graph, we need to provide:

- the name of the graph as key of the Kafka message in the Kafka topic
- the new data for the replace as value of the Kafka message

Thus, we need to modify the producer to create a key-value message. In the Kafka installation directory, execute:

\begin{verbatim}
    bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-replace \
    --property parse.key=true --property key.separator="-
\end{verbatim}

4. To replace the data in the graph, paste this into the same window, and press Enter.

\begin{verbatim}
\end{verbatim}

The key value must be the \texttt{?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:

\begin{verbatim}
    SELECT * WHERE {
        GRAPH ?g {
            ?s ?p ?o
        }
    }
\end{verbatim}

6. The replaced data in the graph should be returned as result.
5.6.3.3 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 - 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, e.g. timestamp information.
- The IRI of the updated document
- The new RDF contents of the updated document

The update is performed as follows:

- The Kafka topic is configured for a specific template.
- The Kafka key of the message holds the value to be used for the ?id parameter in the template’s body - the template binding.
- The Kafka value defines the new data to be added with the update.

**Important:** One SPARQL template typically corresponds to a single document type and is used by a single Kafka sink.

Let’s see how it works.

1. As with the previous update, we already have the Docker container with the Kafka Sink connector running, and will again be using the “kafka-test” repository.

2. With this type of update, we need to first create a SPARQL template for the data update.
   a. In the Workbench, go to Setup  ➤ SPARQL Templates ➤ Create new SPARQL template.
   b. Enter a template IRI (required), e.g., http://example.com/my-template.
   c. As template body, insert:

```
DELETE {
  graph ?g { ?id ?p ?oldValue . }
} INSERT {
  graph ?g { ?id ?p "Successfully updated example" . }
} WHERE {
  graph ?g { ?id ?p ?oldValue . }
}
```

This simple template will look for a given subject in all graphs - ?id, which we will need to supply later when executing the update. The template will then update the object in all triples containing this subject to a new value - "Successfully updated example".

   d. Save it, after which it will appear in the templates list.

Now we 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": "nq", 
"graphdb.template.id": "http://example.com/my-template"}}

Here, we specify the template IRI "graphdb.template.id":"http://example.com/my-template" that we created in the Workbench earlier, which the sink connector will execute.

**Note:** As in the previous example, we have created the connector on a different URL - http://localhost:8085/connectors. If you want to create it on a URL that is already used, you first need to clean the connector that is on it as shown above.

1. To execute a SPARQL update, we need to provide the binding of the template, ?id. It is passed it as the key of the Kafka message, and the data to be add is passed as value.

   In the Kafka installation directory, execute:

   ```
   bin/kafka-console-producer.sh --bootstrap-server localhost:19092 --topic gdb-update \
   --property parse.key=true --property key.separator="-
   ```

2. To execute an update, paste this into the same window, and press Enter.

   ```
   ```

3. To see the updated data, run the query from above in the Workbench SPARQL editor of the “kafka-test” repository:

   ```
   SELECT * WHERE { 
   GRAPH ?g { 
     ?s ?p ?o
   } 
   }
   ```

4. The updated data in the graph should be returned as result.
5.6.4 Configuration properties

The following properties are used to configure the Kafka Sink connector:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Globally unique name to use for this connector.</td>
<td>String</td>
</tr>
<tr>
<td>connector.class</td>
<td>Name or alias of the class for this connector. Must be a sub-class 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.</td>
<td>String</td>
</tr>
<tr>
<td>key.converter</td>
<td>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.</td>
<td>Class</td>
</tr>
<tr>
<td>value.converter</td>
<td>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 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.</td>
<td>Class</td>
</tr>
<tr>
<td>topics</td>
<td>Comma-separated list of topics to consume.</td>
<td>List</td>
</tr>
<tr>
<td>tasks.max</td>
<td>Maximum number of tasks to use for this connector. Default is 1.</td>
<td>Integer</td>
</tr>
<tr>
<td>graphdb.server.url</td>
<td>The URL of the GraphDB server.</td>
<td>String</td>
</tr>
<tr>
<td>graphdb.server.repository</td>
<td>The GraphDB repository where the connector will write the documents.</td>
<td>String</td>
</tr>
<tr>
<td>graphdb.batch.size</td>
<td>The maximum number of documents to be sent from Kafka to GraphDB in one transaction.</td>
<td>Integer</td>
</tr>
<tr>
<td>graphdb.auth.type</td>
<td>The authentication type.</td>
<td>NONE, BASIC, CUSTOM</td>
</tr>
<tr>
<td>graphdb.auth.basic.username</td>
<td>The username for basic authentication.</td>
<td>String</td>
</tr>
<tr>
<td>graphdb.auth.basic.password</td>
<td>The password for basic authentication.</td>
<td>String</td>
</tr>
<tr>
<td>graphdb.auth.header.token</td>
<td>The GraphDB authentication token.</td>
<td>String</td>
</tr>
<tr>
<td>graphdb.update.type</td>
<td>The type of the transaction.</td>
<td>ADD, REPLACE_GRAPH, SMART_UPDATE</td>
</tr>
<tr>
<td>graphdb.update.rdf.format</td>
<td>The format of the documents sent from Kafka to GraphDB. Default is NONE.</td>
<td>RDF, RDFS, XML, OWL, NT, TTL, TTLS, N3, TRIX, TRIG, TRIGS, BRF, NQ, JSON-LD, JSON, JSON-LD, JSONL, RDF, RDF, HTML, HDT</td>
</tr>
<tr>
<td>graphdb.batch.commit.limit.ms</td>
<td>The timeout applied per batch that is not full before it is committed.</td>
<td>Long</td>
</tr>
<tr>
<td>errors.tolerance</td>
<td>Behavior for tolerating errors during connector operation. NONE is the default value and signals that any error will result in an immediate connector task failure; ALL changes the behavior to skip over problematic records.</td>
<td>NONE, ALL</td>
</tr>
<tr>
<td>errors.deadletterqueue.topic.name</td>
<td>The topic name in Kafka brokers to store failed records. Default is blank.</td>
<td>String</td>
</tr>
<tr>
<td>errors.deadletterqueue.topic.replication.factor</td>
<td>Replication factor used to create the dead letter queue topic when it does not already exist. Default is 3.</td>
<td>Short</td>
</tr>
<tr>
<td>errors.retry.timeout</td>
<td>The maximum duration in milliseconds that a failed operation will be reattempted. Default is 0, which means no retries will be attempted. Use -1 for infinite retries.</td>
<td>Long</td>
</tr>
<tr>
<td>errors.retry.timeout.ms</td>
<td>The maximum duration in milliseconds between consecutive retry attempts.</td>
<td>Long</td>
</tr>
</tbody>
</table>

5.6 Kafka Sink Connector
5.7 Reasoning

**Hint:** 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.

### 5.7.1 Logical formalism

GraphDB uses a notation almost identical to R-Entailment defined by Horst. RDFS inference is achieved via a set of axiomatic triples and entailment rules. These rules allow the full set of valid inferences using RDFS semantics to be determined.

Herman ter Horst defines RDFS extensions for more general rule support and a fragment of OWL, which is more expressive than DLP 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 (i.e., 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*.

### 5.7.2 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. This feature must be used with extreme caution because custom rulesets can easily be created, which recursively infer an infinite number of statements making the semantics intractable;
- Variable inequality constraints can be specified in addition to the triple patterns (they can be placed after any premise or consequence). This leads to less complexity compared to R-entailment;
- the *cut* operator can be associated with rule premises. This is an optimization that 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 utilize ‘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.
5.7.3 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, i.e., “/* ... */” for block comments and “//” for end of line comments.

For historic reasons, the way in which terms (variables, URLs and literals) are written differs from Turtle and SPARQL:

- URLs in Prefixes are written without angle brackets
- variables are written without ? or $ and can include multiple alphanumeric chars
- URLs are written in brackets, no matter if they are use prefix or are spelled in full
- datatype URLs are written without brackets, e.g.,

| a <owl:maxQualifiedCardinality> "1"^^xsd:nonNegativeInteger |

See the examples below and be careful when writing terms.

5.7.3.1 Prefixes

This section 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#>
}
```

5.7.3.2 Axioms

This section asserts axiomatic triples, which usually describe the meta-level primitives used for defining the schema such as rdf:type, rdfs:Class, etc. It contains a list of the (variable free) triples, one per line.

For example, the RDF axiomatic triples are defined in the following way:

```
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>
   <rdf:value> <rdf:type> <rdf:Property>
   <rdf:nil> <rdf:type> <rdf:List>
```

5.7. Reasoning
5.7.3.3 Rules

This section 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 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 (see the example below).

Furthermore, inequality constraints can be used to state that the values of the variables in a statement must not be equal to a specific full URI (or its short name), 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 of a rule, then the whole rule will not ‘fire’ if the constraint fails, i.e., 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

The syntax of a rule definition is as follows:

```
Id: <rule_name>
    <premises> <optional_constraints>
    -------------------------------
    <consequences> <optional_constraints>
```

where each premise and consequence is on a separate line.

The following example helps to illustrate 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

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, i.e., 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, e.g., 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 triples patterns, but using different variable names, the extra variant(s) are redundant and better efficiency can be achieved by simply not creating the extra rule variant(s).

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
```

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, i.e., 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>]
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
-----------------------------
```

(continues on next page)
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 if 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>]
-----------------------------
z p y
```/**
```
So, all nodes in the transitive and symmetric chain make relations to all other nodes, i.e., the relation coincides with the Cartesian \( N \times N \), 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 `rdfs plus`. 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, e.g.:

```
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: GraphDB supports multiple named rulesets.
- No downtime deployment: The deployment of new/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(s) 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.

5.7.4 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, 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.
### 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.

<table>
<thead>
<tr>
<th>Ruleset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>No reasoning, i.e., GraphDB operates as a plain RDF store.</td>
</tr>
<tr>
<td>rdfs</td>
<td>Supports the standard model-theoretic RDFS semantics. This includes support for <code>subClassOf</code> and related type inference, as well as <code>subPropertyOf</code>.</td>
</tr>
<tr>
<td>rdfs-plus</td>
<td>Extended version of RDFS with the support also symmetric, inverse and transitive properties, via the OWL vocabulary: <code>owl:SymmetricProperty</code>, <code>owl:inverseOf</code> and <code>owl:TransitiveProperty</code>.</td>
</tr>
<tr>
<td>owl-horst</td>
<td>OWL dialect close to OWL-Horst - essentially pD*.</td>
</tr>
<tr>
<td>owl-max</td>
<td>RDFS and that part of OWL Lite that can be captured in rules (deriving functional and inverse functional properties, all-different, subclass by union/enumeration; min/max cardinality constraints, etc.).</td>
</tr>
<tr>
<td>owl2-ql</td>
<td>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.</td>
</tr>
<tr>
<td>owl2-rl</td>
<td>The OWL2-RL profile - an expressive fragment of OWL2 Full that is amenable for implementation on rule engines.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Conformant behavior</th>
<th>Implemented behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a list of disjoint (data or object) properties and an entity that is related with these properties to objects ( {a, b, c, d, \ldots } ), infer an <code>owl:AllDifferent</code> restriction on an anonymous list of these objects.</td>
<td>For each pair ( {p, q} ) ((p \neq q)) of disjoint (data or object) properties, infer the triple: <code>p owl:propertyDisjointWith q</code> Which is more likely to be useful for query answering.</td>
</tr>
<tr>
<td>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 ).</td>
<td>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.</td>
</tr>
<tr>
<td>If a instance of ( C_1 ), and b instance of ( C_2 ), and ( C_1 ) and ( C_2 ) disjoint, infer: <code>a owl:differentFrom b</code></td>
<td>Impractical for knowledge bases with many members of pairs of disjoint classes, e.g., Wordnet. Instead, this is implemented as a consistency check: If x instance of ( C_1 ) and ( C_2 ), and ( C_1 ) and ( C_2 ) disjoint, then inconsistent.</td>
</tr>
</tbody>
</table>
5.7.4.2 Custom rulesets

GraphDB has an internal rule compiler that can be configured with a custom set of inference rules and axioms. You may define a custom ruleset in a .pie file (e.g., 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 configs/rules folder of the GraphDB distribution.

If the code generation or compilation cannot be completed 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 where the problem is located. Line information is not preserved during the parsing of the rule file.

You must specify the custom ruleset via 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). The compiler is invoked using the mechanism provided by the 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, the class is loaded dynamically and instantiated for further use by GraphDB during inference. The intermediate files are created in the folder that is pointed by the `java.io.tmpdir` system property. The JVM should have sufficient rights to read and write to this directory.

**Note:** Using GraphDB, this is more difficult. It will be necessary to export/backup 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.

5.7.5 Inference

5.7.5.1 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 a java code out of the .pie file contents using the built-in GraphDB rule compiler.
2. Compile the java code (it requires JDK instead of JRE, hence 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.

5.7.5.2 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 checking 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).

5.7.5.3 Retraction of assertions

GraphDB stores explicit and implicit statements, i.e., 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 statement must also be removed.

In the previous versions of GraphDB, this was achieved with a re-computation of the full closure (minimal model), i.e., applying the entailment rules to all explicit statements and computing the inferences. This approach guarantees correctness, but does not scale - the computation is increasingly slow and computationally expensive in proportion to the number of explicit statements and the complexity of the entailment ruleset.

Removal of explicit statements is now achieved in a more efficient manner, by invalidating 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.

The algorithm used in GraphDB works as follows:

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 results (at least one), i.e., this method performs backward chaining.
   • If a result is returned by any query (every rule is represented by a query) in isSupported(), then this statement can be still 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 GOTO 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/RDFS vocabularies) using the `imports repository parameter`.

### Schema update transactions

When fast statement retraction is required, but it is also necessary to update schemas, you can use a special statement pattern. By including an insert for a statement with the following form in the update:

```sparql
```

GraphDB will use the smooth-delete algorithm, but will also traverse read-only statements and allow them to be deleted/inserted. Such transactions are likely to be much more computationally expensive to achieve, but 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:

```sparql
DELETE { 
  # [[schema statements to delete]]
}

INSERT { 
  # [[schema statements to insert]]
}
WHERE { } 
```

### 5.7.6 How To's

#### 5.7.6.1 Operations on rulesets

All examples below use the `sys:` namespace, defined as:

```sparql
prefix sys: <http://www.ontotext.com/owlim/system#>
```
Add a custom ruleset from .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:/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 a built-in ruleset (one of the 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
    "P"refices { a : http://a/ }
  Axioms {}
  Rules {
    Id: custom
    a b c
    a <a:custom1> c
    --
    b <a:custom1> a
  }
}
```
(continues on next page)
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. There is a special kind of statements 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, and remain in the repository forever, even if there are no more nodes related to that particular property or resource. (See Rules optimizations)

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 from “custom” to “test”. Note that “custom” is specified as the subject URI in the default namespace.

**Example**  This renames the ruleset “custom” to “test”.

```plaintext
INSERT DATA {
    _:custom sys:renameRuleset "test"
}
```

Delete a ruleset

The predicate `sys:removeRuleset` deletes the ruleset “test” specified in the object literal.

**Example**

```plaintext
INSERT DATA {
    _:b sys:removeRuleset "test"
}
```

**Note:** Effects on the axiom set when removing a ruleset

When removing a ruleset, we just remove 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**

```plaintext
INSERT DATA {
    _:b sys:consistencyCheckAgainstRuleset "test"
}
```

5.7.6.2 Reinferring

Statements are inferred only when you insert new statements. So, if reconnected to a repository with a different ruleset, it does not take effect immediately. However, you can cause reinference with an Update statement such as:

```plaintext
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.*
5.7.7 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 the implicit statements generated from a specific graph are and the axiomatic triples part of the configured ruleset, i.e., the ones inserted with a special predicate `sys:schemaTransaction`. Find more about it in the plugin’s documentation.

5.8 SHACL Validation

5.8.1 What is SHACL validation?

W3C standard Shapes Constraint Language (SHACL) validation is a valuable tool for efficient data consistency checking, and is supported by GraphDB via RDF4J’s ShaclSail. It is useful in efforts towards data integration, as well as examining data compliance, e.g., every GeoName URI must start with `http://geonames.com/`, or age must be above 18 years.

The language 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. In SHACL, 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 SHACL repository contains the ShaclSail reserved graph `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph`, where all the data is inserted.

5.8.2 Usage

5.8.2.1 Creating and configuring a SHACL repository

A repository with SHACL validation must be created from scratch, i.e., Create new. You cannot modify an already existing repository by enabling the validation afterwards.

Create a repository and enable the Support SHACL validation option. Several additional checkboxes are opened:

* **Cache select nodes**: The ShaclSail retrieves a lot of its relevant data through running SPARQL SELECT queries against the underlying Sail and against the changes in the transaction. This is usually good for performance, but it is recommended to disable this cache while validating large amounts of data as it will be less memory-consuming. Default value is true.
* **Log the executed validation plans**: Logs INFO the executed validation plans as GraphViz DOT. It is recommended 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 is recommended to disable Run parallel validation and Cache select nodes. Default value is false.
* **Validate subjects when target is undefined**: If no target is defined for a NodeShape, that NodeShape will be ignored. Enabling this will make such NodeShapes wildcard shapes and validate...
all subjects. Equivalent to setting sh:targetClass to owl:Thing or rdfs:Resource in an environment with a reasoner. Default value is false.

- **Log validation violations**: Logs (INFO) a list of violations and the triples that caused the violations (BETA). It is recommended 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 is recommended to disable Run parallel validation. Default value is false.

- **RDF4J SHACL extensions**: Activates RDF4J’s SHACL extensions (RSX) that provide additional functionality. 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.

- **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.

Some of these are used for logging and validation - you can find more about it further down in this page.

### 5.8.2.2 Loading shapes and data graphs

You can load shapes using all three key methods for loading data into GraphDB: through the Workbench, with an INSERT query in the SPARQL editor, and through the REST API.

Here is how to do it through the Workbench:

1. Go to User data Import RDF text snippet, and insert the following shape:

```xml
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 ;
] .
```

It indicates that entities of the class Person have a property “age” of the type xsd:integer.

Click Import. In the dialog that opens, select Target graphs Named graph. Insert the ShaclSail reserved graph http://rdf4j.org/schema/rdf4j#SHACLShapeGraph as shown below:
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 (i.e., age is an integer):

   ```prefix  
ex: <http://example.com/ns#>  
sh: <http://www.w3.org/ns/shacl#>  
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, as it is compliant with the shape you just inserted.

   b. Now import incorrect data (i.e., age is a double):

   ```prefix  
ex: <http://example.com/ns#>  
sh: <http://www.w3.org/ns/shacl#>  
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 with all validation violations in both the Workbench and the command line.

5.8.2.3 Deleting shapes and data graphs

There are two ways to delete a SHACL shape: from the GraphDB Workbench and with the RDF4J API.

From the Workbench

1. Go to the SPARQL Editor in the 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 Clean 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 API

Use the following code snippet:

```java
HTTPRepository repository = new HTTPRepository("http://address:port/", "repositoryname");
try {
    RepositoryConnection connection = repository.getConnection();
    connection.begin();
    connection.clear(RDF4J.SHAACL_SHAPE_GRAPH);
    connection.commit();
} catch (Exception e) {
    // handle exception
}
```

5.8.2.4 Updating shapes and data graphs

To successfully update a shape graph, proceed as follows:

1. Go to the SPARQL Editor in the Workbench.
2. Clear the RDF4J graph for storing shapes by running the following update query:

```sparql
CLEAR GRAPH <http://rdf4j.org/schema/rdf4j#SHAACLShapeGraph>
```
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 must be followed.

5.8.2.5 Viewing shapes and data graphs

Currently, shape graphs cannot be accessed with SPARQL inside GraphDB, as 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 code snippet will return all statements inside the shape graph:

```java
HTTPRepository repository = new HTTPRepository("http://address:port/", "repositoryname");
try {
    Model statementsCollector = new LinkedHashModel(connection.getStatements(null, null, null, RDF4J.SHAACL_SHAPE_GRAPH)
        .stream()
        .collect(Collectors.toList()));
} catch (Exception e) {
    // handle exception
}
```

5.8.3 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 the Workbench if it was caused by an update executed in the same Workbench window.

In addition to that, 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**
- **Log validation violations**
- **Log every execution step of the SHACL validation**

All three will log as INFO and appear in the `main-[yyyy-mm-dd].log` file in the `logs` directory of your GraphDB installation.
### 5.8.4 Supported SHACL features

The supported SHACL features are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh:targetClass</td>
<td>Specifies a target class. Each value of <code>sh:targetClass</code> in a shape is an IRI.</td>
</tr>
<tr>
<td>sh:targetNode</td>
<td>Specifies a node target. Each value of <code>sh:targetNode</code> in a shape is either an IRI or a literal.</td>
</tr>
<tr>
<td>sh:targetSubjectsOf</td>
<td>Specifies a subjects-of target in a shape. The values are IRIs.</td>
</tr>
<tr>
<td>sh:targetObjectsOf</td>
<td>Specifies an objects-of target in a shape. The values are IRIs.</td>
</tr>
<tr>
<td>sh:path</td>
<td>Points at the IRI of the property that is being restricted. This may point at a path expression, which would allow you to constrain values that may be several “hops” away from the starting point.</td>
</tr>
<tr>
<td>sh:inversePath</td>
<td>An inverse path is a blank node that is the subject of exactly one triple in a graph. This triple has <code>sh:inversePath</code> as predicate, and the object is a well-formed SHACL property path.</td>
</tr>
<tr>
<td>sh:property</td>
<td>Specifies that each value node has a given property shape.</td>
</tr>
<tr>
<td>sh:or</td>
<td>Specifies the condition that each value node conforms to at least one of the provided shapes.</td>
</tr>
<tr>
<td>sh:and</td>
<td>Specifies the condition that each value node conforms to all provided shapes. This is comparable to conjunction and the logical “and” operator.</td>
</tr>
<tr>
<td>sh:not</td>
<td>Specifies the condition that each value node cannot conform to a given shape. This is comparable to negation and the logical “not” operator.</td>
</tr>
<tr>
<td>sh:minCount</td>
<td>Specifies the minimum number of value nodes that satisfy the condition.</td>
</tr>
<tr>
<td>sh:maxCount</td>
<td>Specifies the maximum number of value nodes that satisfy the condition.</td>
</tr>
<tr>
<td>sh:minLength</td>
<td>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.</td>
</tr>
<tr>
<td>sh:maxLength</td>
<td>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.</td>
</tr>
<tr>
<td>sh:pattern</td>
<td>Specifies a regular expression that each value node matches to satisfy the condition.</td>
</tr>
<tr>
<td>sh:flags</td>
<td>An optional string of flags, interpreted as in SPARQL 1.1 REGEX. The values of <code>sh:flags</code> in a shape are literals with datatype <code>xsd:string</code>.</td>
</tr>
<tr>
<td>sh:nodeKind</td>
<td>Specifies a condition to be satisfied by the RDF node kind of each value node.</td>
</tr>
<tr>
<td>sh:languageIn</td>
<td>Specifies that the allowed language tags for each value node are limited by a given list of language tags.</td>
</tr>
<tr>
<td>sh:datatype</td>
<td>Specifies a condition to be satisfied with regards to the datatype of each value node.</td>
</tr>
<tr>
<td>sh:class</td>
<td>Specifies that each value node is a SHACL instance of a given type.</td>
</tr>
<tr>
<td>sh:in</td>
<td>Specifies the condition that each value node is a member of a provided SHACL list.</td>
</tr>
<tr>
<td>sh:uniqueLang</td>
<td>Can be set to true to specify that no pair of value nodes may use the same language tag.</td>
</tr>
<tr>
<td>sh:minInclusive</td>
<td>Specifies the minimum inclusive value. The values of <code>sh:minInclusive</code> in a shape are literals. A shape has at most one value for <code>sh:minInclusive</code>.</td>
</tr>
<tr>
<td>sh:maxInclusive</td>
<td>Specifies the maximum inclusive value. The values of <code>sh:maxInclusive</code> in a shape are literals. A shape has at most one value for <code>sh:maxInclusive</code>.</td>
</tr>
<tr>
<td>sh:minExclusive</td>
<td>Specifies the minimum exclusive value. The values of <code>sh:minExclusive</code> in a shape are literals. A shape has at most one value for <code>sh:minExclusive</code>.</td>
</tr>
<tr>
<td>sh:maxExclusive</td>
<td>Specifies the maximum exclusive value. The values of <code>sh:maxExclusive</code> in a shape are literals. A shape has at most one value for <code>sh:maxExclusive</code>.</td>
</tr>
<tr>
<td>sh:deactivated</td>
<td>A shape that has the value <code>true</code> for the property <code>sh:deactivated</code> is called deactivated. The value of <code>sh:deactivated</code> in a shape must be either <code>true</code> or <code>false</code>.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh:hasValue</td>
<td>Specifies the condition that at least one value node is equal to the given RDF term.</td>
</tr>
<tr>
<td>dash:hasValueIn</td>
<td>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 &quot;open&quot; like sh:hasValue since it allows values outside of the list.</td>
</tr>
<tr>
<td>sh:target</td>
<td>For use with DASH targets.</td>
</tr>
<tr>
<td>rsx:targetShape</td>
<td>Part of RDF4J's SHACL extensions (RSX) and allows a shape to be the target for your constraints. For more information about the RSX features, see the RSX section.</td>
</tr>
</tbody>
</table>

Implicit sh:targetClass is supported for nodes that are rdfs:Class and either of sh:PropertyShape or sh:NodeShape. Validation for all nodes that are equivalent to owl:Thing in an environment with a reasoner can be enabled by setting setUndefinedTargetValidatesAllSubjects(true).

sh:path is limited to single predicate paths, e.g., ex:age. Sequence paths, alternative paths, inverse paths and the like are not supported.

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.

### 5.9 Virtualization

#### 5.9.1 Overview and features

The data virtualization in GraphDB enables direct access to relational databases with SPARQL queries, which eliminates the need to replicate data. The implementation exposes a virtual SPARQL endpoint, which translates the queries to SQL using a declarative mapping. To achieve this functionality, GraphDB integrates with the open-source Ontop project and extends it with multiple GraphDB specific features.

The following SPARQL features are supported:

- SELECT and CONSTRUCT queries
- Default and named graph triple patterns
- Triple pattern combining: OPTIONAL, UNION, blank node path
- Result filtering and value bindings: FILTER, BIND, VALUES
- Projection modifiers: DISTINCT, LIMIT, ORDER BY
- Aggregates (GROUP BY, SUM, COUNT, AVG, MIN, MAX, GROUP_CONCAT)
- SPARQL functions (STR, IRI, LANG, REGEX)
- SPARQL data type support and their mapping to SQL types
- SUBQUERY

The most common scenario for using data virtualization is when the integrated data is highly dynamic or too big to be replicated. For practical reasons, it is easier to not copy it and accept all limitations like data quality, integrity, and type of supported queries of the underlying information source.

A second 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.

**Note:** The virtual repository has the following specifics:

- it is read-only, meaning that write operations cannot be executed in it;
• COUNT queries cannot be executed;
• sameAs is disabled;
• executing an explain plan is disabled, meaning that graph queries are converted to simple SELECT queries without the graph segment. This will convert a graph query of the type

```sql
SELECT * from <some_graph> WHERE {
}
```

to

```sql
SELECT * WHERE {
}
```

See more about the Ontop framework in its official documentation.

### 5.9.2 Usage scenario

Exposing a virtual endpoint as a repository in GraphDB is done in the following way:

The relational database is loaded in an RDBMS of your choice. After that, a relational database JDBC driver is necessary (e.g., PostgreSQL JDBC driver). It is placed in the `lib` directory of the GraphDB distribution.

Four additional files are needed as well:

1. An OBDA or R2RML file describing the mapping of SPARQL queries to SQL data
2. An OWL file describing the ontology of your data (optional)
3. A properties file for the configuration of the JDBC driver parameters of the following type (here with example values from the sample data we will look at further down in this tutorial):

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

4. A repository config file of the following type, here again with example values (optional):

```xml
@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#> .

<#university-virtual> a rep:Repository;
    rep:repositoryID "university-virtual";
    rep:repositoryImpl [ 
        <http://inf.unibz.it/krdb/obda/quest#obdaFile> "university.obda";
        <http://inf.unibz.it/krdb/obda/quest#owlFile> "university.ttl";
        <http://inf.unibz.it/krdb/obda/quest#propertiesFile> "university.properties";
        <http://inf.unibz.it/krdb/obda/quest#dbMetadataFile> "university-complete-dbmetadata.json";
        rep:repositoryType "graphdb:OntopRepository"
    ];
    rdfs:label "Ontop virtual store with OBDA" .
```

that references the aforementioned OBDA (or R2RML), ontology, and properties files. This file is automatically generated when creating a virtual repository through the Workbench, and is used when creating such a repository via cURL command as described further below.
5. A database metadata file that provides information about uniqueness and non-null constraints for database tables and views (optional).

These files are used to create a virtual repository in GraphDB, in which you can then query the relational database. Let’s consider the following relational database containing university data.

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

uni1.student

<table>
<thead>
<tr>
<th>s_id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>Doe</td>
</tr>
</tbody>
</table>

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

uni1.academic

<table>
<thead>
<tr>
<th>a_id</th>
<th>first_name</th>
<th>last_name</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Anna</td>
<td>Chambers</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Edward</td>
<td>May</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Rachel</td>
<td>Ward</td>
<td>8</td>
</tr>
</tbody>
</table>

Similarly, this table contains the local ID, 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

uni1.course

<table>
<thead>
<tr>
<th>c_id</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>Linear Algebra</td>
</tr>
</tbody>
</table>

This table contains the local ID and the title of the courses. The column c_id is a primary key.

uni1.teaching

<table>
<thead>
<tr>
<th>c_id</th>
<th>a_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1</td>
</tr>
<tr>
<td>1234</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

uni1.course-registration

<table>
<thead>
<tr>
<th>c_id</th>
<th>s_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>1</td>
</tr>
<tr>
<td>1234</td>
<td>2</td>
</tr>
</tbody>
</table>
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.

5.9.3 Setup and configuration

5.9.3.1 JDBC driver

As mentioned above, in order to create a virtual repository in GraphDB, you need to first install a JDBC driver for your respective relational database.

Place the driver .jar file in the lib directory of the GraphDB distribution. In case you are using GraphDB from a native installation, the driver file name should be jdbc-driver.jar. Restart GraphDB if it is running.

5.9.3.2 Configuration files

Before creating a virtual repository, you will need the following files (available for download below):

- a properties file with the JDBC configuration parameters
- an OBDA mapping file describing the mapping of SPARQL queries to SQL data
- an OWL ontology file describing the ontology of your data (optional)
- a DB metadata file providing information about uniqueness and non-null constraints for database tables and views (optional)

5.9.3.3 Creating a virtual repository from the Workbench

With generic JDBC driver

1. When creating a repository from the Workbench, select the Ontop option.

2. GraphDB supports several database JDBC drivers. When creating an Ontop repository, the default setting is Generic JDBC Driver. This means that you need to configure and upload your own JDBC properties file (available as a template for download above).

3. In the fields for JDBC properties file and OBDA or R2RML file, upload the corresponding files. The Ontology, Constraint, and DB metadata files are optional.

4. You can also test the connection to your SQL database with the button on the right.

5. Click Create.
Note: Once you have created an Ontop repository, its type cannot be changed.

With one of the other supported database drivers

For ease of use, GraphDB also supports drivers for five other commonly used databases integrated into the Ontop framework: MySQL, PostgreSQL, Oracle, MS SQL Server, and DB2. Selecting one of them offers the advantage of not having to configure the JDBC properties file yourself, as its Driver class and URL property values are generated by GraphDB.

To use one of these database drivers:

1. Select the type of SQL database you want to use from the drop-down menu.
2. Download the corresponding driver by clicking the Download JDBC driver link on the right of the Driver class field, place it in the lib directory of the GraphDB distribution, and restart GraphDB if it is running.
3. Fill in the required fields for each driver (Hostname, Database name, etc.).
4. Upload the OBDA/R2RML file. (The Ontology, Constraint, and DB metadata files are optional, just as with the generic JDBC driver)
5. You can also test the connection to your SQL database with the button on the right.
6. Click Create.
5.9.3.4 Creating a virtual repository using cURL

To create a virtual repository via the API, you need the following files described above, all placed in the same directory (here, we are using the universities examples again):

- **repo-config.ttl**: the config file for the repository
- **university.properties**: the JDBC properties file
- **university.obda**: the OBDA/R2RML file

As mentioned earlier, the OWL ontology, constraint, and DB metadata files are optional.

Execute the following cURL command (here including the DB metadata file):

```
curl -X POST http://localhost:7200/rest/repositories -H 'Content-Type: multipart/form-data' -F "config=@repo-config.ttl" -F "obdaFile=@university.obda" -F "propertiesFile=@university.properties" -F "dbMetadataFile=@university-complete-db-metadata.json"
```

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

5.9.4 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 known as OBDA, which is much shorter and easier to learn, and supports an automatic bidirectional transformation to R2RML.

Mappings represent OWL assertions: one set of OWL assertions for each result row is returned by the SQL query in the mapping. The assertions are those that are obtained by replacing the placeholders with the values from the relational database.

Mappings consist of:

- **source**: a SQL query that retrieves some data from the database
- **target**: a form of template that indicates how to generate OWL assertions in a Turtle-like syntax.

All examples in this documentation use the internal OBDA mapping language.

Let’s map the uni1-student table using an OBDA template.

The information source is the following:

```
SELECT *
FROM "uni1"."student"
```

And the target mapping file is:

```
ex:uni1/student/{s_id} a :Student ;
  foaf:firstName {first_name}^^xsd:string ;
  foaf:lastName {last_name}^^xsd:string .
```

The target part is described using a Turtle-like syntax while the source part is a regular SQL query.

We used the primary key s_id to create the URI. This practice enables Ontop to remove self-joins, which is very important for optimizing the query performance.

This entry could be split into three mapping assertions:

```
ex:uni1/student/{s_id} a :Student .
ex:uni1/student/{s_id} foaf:firstName {first_name}^^xsd:string .
ex:uni1/student/{s_id} foaf:lastName {last_name}^^xsd:string .
```
Mapping the `uni1-course` table would look as follows:

The source will be:

```sql
SELECT *
FROM "uni1"."course"
```

And the target:

```reasoning
ex:uni1/course/{c_id} a :Course ;
  :title {title} ;
  :isGivenAt ex:uni1/university .
```

5.9.5 SPARQL endpoint

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:

   ```reasoning
   PREFIX voc: <http://example.org/voc#>
   SELECT ?p
   WHERE {
     ?p a voc:FacultyMember .
   }
   ```

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

   ```reasoning
   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:

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

   SELECT ?title ?fName ?lName
   { ?teacher rdf:type voc:Professor .
     ?teacher foaf:lastName ?lName .
     OPTIONAL {
       ?teacher foaf:firstName ?fName .
     }
   }
   ```
5.9.6 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.

Let’s see how this 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 that give courses that are not held at university2:

   ```sparql
   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:ontop_repo> {
       SELECT DISTINCT ?person ?firstName ?lastName
       WHERE {
         ?person foaf:firstName ?firstName ;
         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:

   ```sparql
   PREFIX voc: <http://example.org/voc#>
   SELECT ?firstName ?lastName
   WHERE {
     ?teacherId a voc:UniversityTeacher;
     voc:firstName ?firstName;
     voc:lastName ?lastName;
   } LIMIT 200
   ```

   Result:

<table>
<thead>
<tr>
<th>Teacher ID</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steve</td>
<td>Johnson</td>
</tr>
<tr>
<td>2</td>
<td>Rachel</td>
<td>Smith</td>
</tr>
<tr>
<td>3</td>
<td>Karen</td>
<td>Williams</td>
</tr>
<tr>
<td>4</td>
<td>Anna</td>
<td>Chambers</td>
</tr>
<tr>
<td>5</td>
<td>Edward</td>
<td>Pope</td>
</tr>
</tbody>
</table>

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 our demonstration, this query filters them by firstName that contains the letter “a”.

5.9. Virtualization
PREFIX voc: <http://example.org/voc#>
select * where {
    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>
<thead>
<tr>
<th>teacherId</th>
<th>institution</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.org/voc/knowledge/1">http://example.org/voc/knowledge/1</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Anna&quot;</td>
<td>&quot;Chambers&quot;</td>
</tr>
<tr>
<td><a href="http://example.org/voc/knowledge/2">http://example.org/voc/knowledge/2</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Gerard&quot;</td>
<td>&quot;Cosme&quot;</td>
</tr>
<tr>
<td><a href="http://example.org/voc/knowledge/3">http://example.org/voc/knowledge/3</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Edward&quot;</td>
<td>&quot;May&quot;</td>
</tr>
<tr>
<td><a href="http://example.org/voc/knowledge/4">http://example.org/voc/knowledge/4</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Tanya&quot;</td>
<td>&quot;Abby&quot;</td>
</tr>
<tr>
<td><a href="http://example.org/voc/knowledge/5">http://example.org/voc/knowledge/5</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Anna&quot;</td>
<td>&quot;Chambers&quot;</td>
</tr>
<tr>
<td><a href="http://example.org/voc/knowledge/6">http://example.org/voc/knowledge/6</a></td>
<td><a href="http://example.org/voc/knowledge/university">http://example.org/voc/knowledge/university</a></td>
<td>&quot;Mache&quot;</td>
<td>&quot;Ward&quot;</td>
</tr>
</tbody>
</table>

5.9.7 Limitations

Data virtualization also 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 in databases X and Y, we 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.

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 perform auto-suggest/auto-complete type of indexes nor graph traversals or inferencing.

5.10 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.

The REST API calls fall into these major categories:

5.10.1 Cluster group controller

**Note:** This feature requires a GraphDB Enterprise license.

Use the cluster group controller 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 these cURL examples for cluster group management.
5.10.2 Data import

Use the data import API to import data in GraphDB. You can choose between server files and a remote URL.
See these cURL examples for data import.

5.10.3 Location management

Use the location management API to attach, edit, or detach locations.
See these cURL examples for location management.

5.10.4 Recovery management

Use the backup/restore management API to create, edit, and delete backups, as well as to restore a GraphDB instance from a backup.
See these cURL examples for recovery management.

5.10.5 Repository management

Use the repository management API to add, edit, or remove a repository to/from any attached location. Unlike the RDF4J API, you can work with multiple remote locations from a single access point. When combined with the location management, it can be used to automate the creation of multiple repositories across your network.
See these cURL examples for repository management.

5.10.6 Saved queries

Use the saved queries API to create, edit or remove saved queries. It is a convenient way to automate the creation of saved queries that are important to your project.
See these cURL examples for saved queries.

5.10.7 Security management

Use the security management API to enable or disable security and free access, as well as add, edit, or remove users, thus integrating the Workbench security into an existing system.
See these cURL examples for security management.

5.10.8 SPARQL templates

Use the SPARQL template management API to create, edit, delete, and execute SPARQL templates, as well as to view all templates and their configuration.
See these cURL examples for SPARQL template management.
5.10.9 SQL views management

Use the SQL views management API to access, create, and edit SQL views (tables), as well as to delete existing saved queries and view all SQL views for the active repository.

See these *cURL examples for SQL views management.*

5.10.10 Authentication

Use this login REST API endpoint to obtain a GDB token in exchange for username and password.

See this *cURL example for authentication.*

You can find more information about each REST API functionality group and its operations in *Help REST API Documentation*, as well as execute them directly from there and see the results.

**GraphDB Workbench API**

```
click on a functionality group to expand it and see the operations that it includes. Click on an operation to see details about it.
```

---

230 Chapter 5. Usage
5.11 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 comprises 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.

5.11.1 RDF4J API

Programmatically, GraphDB can be used via the RDF4J Java framework of classes and interfaces. Documentation for these interfaces (including Javadoc). Code snippets in the sections below are taken from, or are variations of, the developer-getting-started examples that come with the GraphDB distribution.

5.11.1.1 Accessing a local repository

With RDF4J 2, repository configurations are represented 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, etc. A short repository configuration is taken from the developer-getting-started template file repo-defaults.ttl.

```text
@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/trree/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/" ;
                                      graphdb:repository-type "file-repository" ;
                                      graphdb:imports "/ontology/owl.rdfs" ;
                                      graphdb:defaultNS "http://example.org/" .
                        ] ].
```

The Java code that uses the configuration to instantiate a repository and get a connection to it is as follows:

```java
// 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();
```

(continues on next page)
The procedure is as follows:

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 interact with the repository, e.g., executing queries, adding statements, committing transactions, counting statements, etc. See the developer-getting-started examples.

Note: Example above assumes that GraphDB Free edition is used. If using Standard or Enterprise editions, a valid license file should be set to the system property `graphdb.license.file`.

### 5.11.1.2 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, e.g., Tomcat, and allows client applications to interact with repositories located on remote machines. In order to connect to and use a remote repository, you have to replace the local repository manager with a remote one. The URL of the RDF4J server must be provided, but no repository configuration is needed 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:

```java
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
5.11.2 SPARQL endpoint

The RDF4J HTTP server is a fully fledged SPARQL endpoint - the RDF4J HTTP protocol is a superset of the SPARQL 1.1 protocol. It provides an interface for transmitting SPARQL queries and updates to a SPARQL processing service and returning the results via HTTP to the entity that requested them.

Any tools or utilities designed to interoperate with the SPARQL protocol will function with GraphDB because it exposes a SPARQL-compliant endpoint.

5.11.3 Graph Store HTTP Protocol

The Graph Store HTTP Protocol is fully supported 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.

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.

5.12 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 BI tool of your choice (e.g., Tableau or Microsoft Power BI). This capability is provided by GraphDB’s JDBC driver, which enables you to create SQL views using SPARQL SELECT queries, and to access all GraphDB features including plugins and SPARQL federation. The functionality 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. For simplicity of the table creation process, we have integrated the SQL View Manager in the GraphDB Workbench. It allows you to 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:** Over this functionality, you can only read data from the repository. Write operations are not enabled.

5.12.1 Configuration

5.12.1.1 Prerequisites

You need to download the GraphDB JDBC driver (graphdb-jdbc-remote-10.0.0.jar), a self-contained .jar file. The driver needs to be installed according to the requirements of the software that supports JDBC. See below for specific instructions.

For the purposes of this guide, we will be using the Netherlands restaurants RDF dataset. Upload it into a GraphDB repository, name it nl_restaurants, and set it as the active repository.

Now, let’s access its data over the JDBC driver.
5.12.1.2 Creating a SQL view

1. Go to Setup JDBC. Initially, the list of SQL table configurations will be empty as none are configured.
2. Click Create new SQL table configuration.

In the view that opens, there are two tabs:

- **Data query**: The editor where to input 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 two columns using `rdfs:label`-`id` and `label`.

```
PREFIX <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?id ?label {
  FILTER (?id = 'restaurant_data')
}
```

**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 would not recognize it.

- **Column types**: Here, you can configure 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.

  Note that in order to create a table, it must contain at least one column.

3. Fill in a Table name for your table, e.g., `restaurant_data`. This field is mandatory and 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:
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. Additionally, SQL types will be suggested 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 (e.g., decimal).
   - 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. It does not need to be saved in order to be previewed.
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 further below, let’s also create another SQL view with the following query:

```sql
PREFIX ex:<http://example.com/ex>
PREFIX base:<http://example/base/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

  ?s a base:Restaurant;
  rdfs:label ?restaurant_name;
  ex:inCity ?city_id;
  ex:address ?address;
  ex:zipcode ?zipcode;
  ex:latitude ?latitude;
  ex:longitude ?longitude.

  ?country_id rdfs:label ?country.
} # !filter
```

Name it `restaurant_location` and save it.
5.12.1.3 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, with the only difference 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.

5.12.1.4 Deleting a SQL view

To delete a SQL view, click the delete icon next to its name in the available SQL views list.

5.12.2 Type mapping

This table shows all RDF data types, their type equivalent in SQL, and the conversion (or mapping) of RDF to SQL values.

<table>
<thead>
<tr>
<th>Metadata type</th>
<th>SQL type</th>
<th>Default precision and scale</th>
<th>RDF to SQL</th>
<th>Default RDF type in FILTER()</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>VARCHAR</td>
<td>1,000</td>
<td>Literal.stringValue()</td>
<td>plain literal or literal with language tag</td>
</tr>
<tr>
<td>IRI</td>
<td>VARCHAR</td>
<td>500</td>
<td>IRI.stringValue()</td>
<td>IRI</td>
</tr>
<tr>
<td>boolean</td>
<td>BOOLEAN</td>
<td></td>
<td>Literal.booleanValue()</td>
<td>literal with xsd:boolean</td>
</tr>
<tr>
<td>byte</td>
<td>BYTE</td>
<td></td>
<td>Literal.byteValue()</td>
<td>literal with xsd:byte</td>
</tr>
<tr>
<td>short</td>
<td>SHORT</td>
<td></td>
<td>Literal.shortValue()</td>
<td>literal with xsd:short</td>
</tr>
<tr>
<td>int</td>
<td>INT</td>
<td></td>
<td>Literal.intValue()</td>
<td>literal with xsd:int</td>
</tr>
<tr>
<td>long</td>
<td>LONG</td>
<td></td>
<td>Literal.longValue()</td>
<td>literal with xsd:long</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT</td>
<td></td>
<td>Literal.floatValue()</td>
<td>literal with xsd:float</td>
</tr>
<tr>
<td>double</td>
<td>DOUBLE</td>
<td></td>
<td>Literal.doubleValue()</td>
<td>literal with xsd:double</td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
<td>19, 0</td>
<td>Literal.decimalValue()</td>
<td>literal with xsd:decimal</td>
</tr>
<tr>
<td>date</td>
<td>DATE</td>
<td></td>
<td>See below</td>
<td>literal with xsd:date, no timezone</td>
</tr>
<tr>
<td>time</td>
<td>TIME</td>
<td></td>
<td>See below</td>
<td>literal with xsd:time, no timezone</td>
</tr>
<tr>
<td>timestamp</td>
<td>TIMESTAMP</td>
<td></td>
<td>See below</td>
<td>literal with xsd:datetime, no timezone</td>
</tr>
</tbody>
</table>

Each metadata type may be followed by optional precision and scale in parentheses, e.g., 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 (e.g., “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 SQL WHERE is pushed to a SPARQL FILTER().

Dates, times, and timestamps are tricky as there is no timezone support in those types in SQL. 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 we consider “no timezone” to be equivalent to “Z” (i.e., +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.

5.12.3 WHERE to FILTER conversion

The following SQL operators are converted to FILTER 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.

5.12.4 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 the Workbench is functioning properly.

After installing it, execute the following steps:

1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-10.0.0.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 window, 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, e.g., 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 again click the + to create a new one.

9. You will see the newly created driver and its URL visible in the dialog window. Choose a name for the alias, e.g., 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 hit Ctrl+Enter (or the Run SQL icon above):
5.12.5 Usage examples

5.12.5.1 Tableau

Now let’s transform your RDF data into SQL:

1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-10.0.0.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, in our case 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 centre 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:

5.12.5.2 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, e.g., Easysoft’s ODBC-JDBC Gateway.

After downloading and installing the gateway in your Windows operating system, connect it to GraphDB the following way:

1. Download the GraphDB JDBC driver (graphdb-jdbc-remote-10.0.0.jar).
2. From the main menu, go to *ODBC Data Sources (64-bit)*.
3. In the dialog window, 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, we will configure the connection to GraphDB:
in the DSN field, enter the name of the new driver, for example “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 window, 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 window, 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.

5.12.6 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:

```sparql
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;
}
```

(continues on next page)
It is generated as an .rq file upon creation of a SQL table from the 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

5.13 Plugins

5.13.1 Activate & enable plugins

5.13.1.1 Activate/deactivate plugins

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

From the Workbench

For most of the plugins, this can be done from the Workbench in Setup Plugins. By default, all plugins available there are activated.

<table>
<thead>
<tr>
<th>Plugins for repository my_repo</th>
<th>autocomplete</th>
<th>dependencies-plugin</th>
<th>elastic-search-connector</th>
<th>expose-entity</th>
<th>geospatial</th>
<th>kafka-connector</th>
<th>magenta</th>
<th>notifications</th>
<th>proof</th>
<th>script</th>
<th>similarity</th>
<th>sparql-rmm</th>
<th>text-mining</th>
<th>url</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

   ![Autocomplete Plugin is not active for this repository. When a plugin is not active it cannot be used or configured.]
   
   ![Activate]

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

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

   To deactivate it:

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

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

   **List plugins and their state**

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

   ```sparql
   ```

   **5.13.1.2 Enable/disable plugins**

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

   **Autocomplete index**

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

   **Change tracking**

   You can enable change tracking for a certain transaction ID. See how to do it here.
Data history & versioning

See how to enable the plugin here.

GeoSPARQL support

See how to enable the plugin here.

RDF Rank

This refers to the RDF Rank filtered mode when you want to calculate the rank only of certain entities. See how to do it here.

5.13.2 Plugin API

5.13.2.1 What is the 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/removal of plugins from the system without having to recompile GraphDB or change any configuration files.

5.13.2.2 Description of a 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 Java package, i.e., com.ontotext.trree.sdk. Here is what the plugin interface looks like in an abbreviated form:

```java
/**
 * The base interface for a GraphDB plugin. As a minimum a plugin must implement this interface.
 * Plugins also need to be listed in META-INF/services/com.ontotext.trree.sdk.Plugin so that Java's
 * 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);

    /**
     * Plugin initialization method called once when the repository is being initialized, after the plugin
     * has been
     */
```

(continues on next page)
As 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 has to be written on a separate line in such a file.

The only method introduced by the Service interface is getName(), which provides the plugin’s (service’s) name. This name must be unique within a particular GraphDB repository, and serves as a plugin identifier that can be used at any time to retrieve a reference to the plugin instance.
Plugins need not implement this interface directly but rather implement \{@link Plugin\}.

```java
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.

5.13.2.3 The life cycle of a plugin

A plugin’s life cycle consists of several phases:

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.rrree.sdk.Plugins service registry files, and 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 it, 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 it gets the chance to do whatever initialization work it needs to do. 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 IRIIs 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 till 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, the addition and removal of statements.

Shutdown

During repository shutdown, each plugin is prompted to execute its own shutdown routines, free resources, flush data to disk, etc. This must be done in the `shutdown(ShutdownReason reason)` method.

This phase is also called when a plugin is disabled after repository initialization.

5.13.2.4 Repository internals

The repository internals are accessed via an instance of `PluginConnection`:

```java
/**
 * 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();
} (continues on next page)
```
public GraphDB() {
    Statements getStatements();
    long getTransactionId();
    boolean isTesting();
    SystemProperties getProperties();
    String getFingerprint();
}

5.13. Plugins
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 and various system and repository properties (SystemProperties).

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 type and properties, resolving entities, etc.

It is possible to declare two RDF objects to be equivalent in a GraphDB repository, e.g., 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 have to 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 ‘statement’ means a quadruple of subject, predicate, object, and context RDF entity identifiers. Statements can be searched for but not modified.
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. Furthermore, some properties of each statement are 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).

Here is a brief example that puts `Statements`, `Entities`, and `StatementIterator` together in order to output all literals that are related to a given URI:

```java
// 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()) {
    // 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 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.

### System properties

`PluginConnection` provides access to various static repository and system properties via `getProperties()`. The values of these properties are set at repository initialization time and will not change while the repository is operating.

The `getProperties()` method returns an instance of `SystemProperties`:

```java
/**
 * 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.
     */
    (continues on next page)
```
* @return true if read-only, false otherwise
*/
boolean isReadOnly();

/**
 * Returns the number of bits needed to represent an entity id
 */
int getEntityIdSize();

/**
 * Returns the type of the current repository.
 */
RepositoryType getRepositoryType();

/**
 * Returns the full GraphDB version string.
 */
String getVersion();

/**
 * Returns the GraphDB major version component.
 */
int getVersionMajor();

/**
 * Returns the GraphDB minor version component.
 */
int getVersionMinor();

/**
 * Returns the GraphDB patch version component.
 */
int getVersionPatch();

/**
 * Returns the number of cores in the currently set license up to the physical number of cores on the
 * machine.
 */
int getNumberOfLicensedCores();

/**
 * The possible editions for GraphDB repositories.
 */
enum RepositoryType {
    /**
     * GraphDB Free repository
     */
    FREE,

...
Repository properties

There are some dynamic repository properties that may change once a repository has been initialized. These properties are:

**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`.

5.13.2.5 Query processing

As already mentioned, 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 be participating in request pre-processing must implement the `Preprocessor` interface. It looks like this:

```java
public interface Preprocessor {

  /**
   * Interface that should be implemented by all plugins that need to maintain per-query context.
   */

  public 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, initializes, 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 it is, 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.

5.13. Plugins
e.g., 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 are not able to 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. In fact, most plugins need to participate in exactly this phase. This is the point where request statement patterns need to get evaluated and statement results are returned.

For example, consider the following SPARQL query:

```sparql
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, i.e., the ones persisted on the disk.

Here is the PatternInterpreter interface:

```java
/**
 * 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 Entities#BOUND or Entities#UNBOUND)
     * @param predicate predicate ID (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param object object ID (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param context context value (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param pluginConnection an instance of PluginConnection
     * @param requestContext context object as returned by 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,
                     RequestContext requestContext);

    /**
     * Interpret basic triple pattern and return StatementIterator with results
     *
     * @param subject subject ID (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param predicate predicate ID (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param object object ID (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param context context value (alternatively Entities#BOUND or Entities#UNBOUND)
     * @param pluginConnection an instance of PluginConnection
     * @param requestContext context object as returned by Preprocessor.preprocess() or null
     * @return statement iterator of results
     */
    default StatementIterator interpret(long subject, long predicate, long object, long context, PluginConnection pluginConnection,
                                         RequestContext requestContext);```

(continues on next page)
The `estimate()` and `interpret()` methods take the same arguments and are used in the following way:

- Given a statement pattern (e.g., 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 on 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 (e.g., `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:

```java
<@
* Interface implemented by plugins that want to interpret list-like triple patterns
*
public interface ListPatternInterpreter {

(continues on next page)
```
/**
 * 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 \{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 \{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 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 \{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 \texttt{PatternInterpreter} by having multiple objects passed as an array of \texttt{long}, instead of a single \texttt{long} object. The semantics of both methods is equivalent to the one in the basic pattern interpretation case.

\texttt{SubjectListPatternInterpreter} handles lists in the \textit{subject} position:

/*
 * Interface implemented by plugins that want to interpret list-like triple patterns
 */

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 \{Preprocessor.preprocess()\} or null
 * @return approximate number of results that could potentially be returned for this parameters by the
 */

(continues on next page)
* **interpret()** method

```java
public 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 is 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:

```java
/**
 * 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);

    /**
     * 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 object as returned by \{code 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
     */
}
```
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 having post-processing functionality 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.

5.13.2.6 Update processing

Updates involving specific predicates

As well as query/read processing, plugins are able to process update operations for statement patterns containing specific predicates. In order to intercept updates, a plugin must implement the `UpdateInterpreter` interface. During initialization, the `getPredicatesToListenFor()` is called once by the framework, 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.
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 (i.e., delete everything in the repository) the operation will be optimized internally and individual events will not be fired. This means that \texttt{UpdateInterpreter} and \texttt{StatementListener} will not be called.

\texttt{ClearInterpreter} is an interface that allows plugins to detect the removal of entire contexts or removal of all data in the repository:

```java
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 PluginConnection
   */
  void beforeClear(long context, PluginConnection pluginConnection);

  /**
   * Notification called after the statements have been removed from the given context.
   *
   * @param context the ID of the context or 0 if all contexts
   * @param pluginConnection an instance of 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:

```java
/**
 * 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.
5.13.2.7 Transactions

A plugin may require to participate in the transaction workflow, e.g., because the plugin 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:

```java
public interface PluginTransactionListener {
    // The (PluginTransactionListener) allows plugins to be notified about transactions (start, commit+completed or abort)

    /**
     * Notifies the listener about the start of a transaction.
     * @param pluginConnection an instance of 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 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 (PluginTransactionCommit). Plugins are encouraged to perform such an abort
     * after the commit phase to ensure a complete transaction.
     * @param pluginConnection an instance of PluginConnection
     */
    void transactionCompleted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the abortion of a transaction. This will be the last event in an aborted transaction.
     * Plugins should revert any modifications caused by this transaction, including the fingerprint.
     * @param pluginConnection an instance of 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.
     * 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 `transactionCommit`) running in the main thread.
     */
}
```

(continues on next page)
Each transaction has a beginning signalled by a call to `transactionStarted()`. Then the transaction can proceed in several ways:

- **Commit and completion:**
  - `transactionCommit()` is called;
  - `transactionCompleted()` is called.
- **Commit followed by abortion (typically because another plugin aborted the transaction in its own `transactionCommit()`):**
  - `transactionCommit()` is called;
  - `transactionAborted()` is called.
- **Abortion before entering commit:**
  - `transactionAborted()` is called.

Plugins should strive to do all heavy transaction work in `transactionCommit()` in such a way that call to `transactionAborted()` can revert the changes. Plugins may throw exceptions in `transactionCommit()` in order to abort the transaction, e.g., if some constraint was violated.

Plugins should do no heavy processing in `transactionCompleted()` and are not allowed to throw exceptions there. Such exceptions will be logged and ignored, and the transaction will still go through normally.

The `transactionAbortedByUser()` will be called asynchronously (e.g., while the plugin is executing `transactionCommit()` in the main update thread) when a user requests the transaction to be aborted. The plugin may use this to signal its other thread to abort processing at earliest convenience or simply ignore the request.

### 5.13.2.8 Exceptions

Plugins may throw exceptions on invalid input, constraint violations or unexpected events (e.g. 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.
5.13.2.9 Accessing other plugins

Plugins can make use of the functionality of 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 facilitate query result scoring and ordering. This is not a matter of re-using program code (e.g., in a .jar with common classes), but rather 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 \texttt{PluginDependency} interface:

```java
/**
 * 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 \texttt{PluginLocator} instance in the plugin.
     *
     * @param locator a \texttt{PluginLocator} instance
     */
    void setLocator(PluginLocator locator);
}
```

They are then injected into an instance of the \texttt{PluginLocator} interface (during the configuration phase), which does the actual plugin discovery for them:

```java
/**
 * Interface that supports obtaining of a plugin instance by plugin name. An object implementing this
 * interface is injected into plugins that implement the \texttt{PluginDependency} interface.
 */
public interface PluginLocator {
    /**
     * Retrieves a \texttt{Plugin} instance by plugin name
     *
     * @param name name of the plugin
     * @return a \texttt{Plugin} instance or null if a plugin with that name is not available
     */
    Plugin locate(String name);

    /**
     * Retrieves a \texttt{RDFRankProvider} instance.
     *
     * @return a \texttt{RDFRankProvider} instance or null if no \texttt{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 \texttt{RDFRankProvider} interface. The sole implementation is the RDF Rank plugin but it can be easily replaced by another implementation. By having a dedicated interface it is easy for plugins to get access to RDF ranks without relying on a specific implementation.
5.13.2.10 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 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.

It is suitable for plugins that are unimportant for query results or update executions, e.g., 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 will be called only 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.
This can be used to implement virtual contexts and 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 extensions 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 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 able to provide 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 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, e.g. invalid input. Extends PluginException.

ServerErrorException Server error, e.g. something unexpected such as lack of disk permissions. Extends PluginException.

5.13.2.11 Adding external plugins to GraphDB

With the graphdb.extra.plugins property, you can attach a directory with external plugins when starting GraphDB. It is set the following way:

```
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.
5.13.2.12 Putting it all together: example plugins

A project containing two example plugins, `ExampleBasicPlugin` and `ExamplePlugin` can be found here.

**ExampleBasicPlugin**

`ExampleBasicPlugin` has the following functionality:

- It interprets the pattern `?s <http://example.com/now> ?o` and binds the object to a literal containing the system date/time of the machine running GraphDB. The subject position is not used and its value does not matter.

The plugin implements the `PatternInterpreter` interface. A date/time literal is created 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:

```java
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 not to forget 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:

```java
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
        // (continues on next page)
    }
}
```

5.13. Plugins
SystemDate only when we detect the `<http://example.com/time>` predicate.

```java
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);
```}

```java
@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;
}
```

```java
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** has the following functionality:

- 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/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/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()) containing 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:

```java
public class ExamplePlugin extends PluginBase implements Preprocessor, Postprocessor {

    // Preprocessor interface methods
    @Override
    public RequestContext preprocess(Request request) {
        // We are interested only in QueryRequests
        if (request instanceof QueryRequest) {
            QueryRequest queryRequest = (QueryRequest) request;

            // Implement the plugin logic here...
        }

        return request;
    }

    // ... initialize() and getName()
}
```

(continues on next 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
    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 (i.e., if the FROM clause was detected by preprocess()). Finally, flush() returns a singleton iterator, containing the desired binding set in the required case or 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 via implementing UpdateInterpreter:

```java
public class ExamplePlugin extends PluginBase implements UpdateInterpreter, Preprocessor, Postprocessor {
    // ... initialize() and getName()

    // 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.getEntity().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 the plugin is interested in via 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 to signal that the plugin has processed the update and it needs not be inserted as regular data.
5.13.3 RDF Rank

5.13.3.1 What is RDF Rank

RDF Rank is an algorithm that identifies the more important or more popular entities in the repository by examining their interconnectedness. The popularity of entities can then be used to order the query results in a similar way to the internet search engines, the way Google orders search results using PageRank.

The RDF Rank component computes a numerical weighting for all nodes in the entire RDF graph stored in the repository, including URIs, blank nodes, literals, and RDF-star (formerly RDF*) 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/popularity.

Since the values range from 0 to 1, the weights can be used for sorting 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 for sorting results by their popularity:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT
WHERE {
  ?Person a opencyc-en:Entertainer .
  ?Person rank:hasRDFRank ?rank .
} ORDER BY DESC(?rank) LIMIT 100
```

As seen in the example query, 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 0.01. You can as well retrieve the rank with precision of 0.001, 0.0001 and 0.00001 using respectively rank#hasRDFRank3, rank#hasRDFRank4, and rank#hasRDFRank5.
In order to use this mechanism, the RDF ranks for the whole repository must be computed in advance. This is done by committing a series of SPARQL updates that use special vocabulary to parameterize the weighting algorithm, followed by an update that triggers the computation itself.

5.13.3.2 Parameters

RDF Rank is fully controllable from Setup → RDF Rank.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Maximum iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><a href="http://www.ontotext.com/owlim/RDFRank#maxIterations">http://www.ontotext.com/owlim/RDFRank#maxIterations</a></td>
</tr>
<tr>
<td>Description</td>
<td>Sets the maximum number of iterations of the algorithm over all entities in the repository.</td>
</tr>
<tr>
<td>Default</td>
<td>20</td>
</tr>
<tr>
<td>Example</td>
<td>PREFIX rank: <a href="http://www.ontotext.com/owlim/RDFRank#">http://www.ontotext.com/owlim/RDFRank#</a> INSERT DATA ( rank:maxIterations rank:setParam &quot;16&quot; . )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><a href="http://www.ontotext.com/owlim/RDFRank#epsilon">http://www.ontotext.com/owlim/RDFRank#epsilon</a></td>
</tr>
<tr>
<td>Description</td>
<td>Terminates the weighting algorithm early when the total change of all RDF Rank scores has fallen below this value.</td>
</tr>
<tr>
<td>Default</td>
<td>0.01</td>
</tr>
<tr>
<td>Example</td>
<td>PREFIX rank: <a href="http://www.ontotext.com/owlim/RDFRank#">http://www.ontotext.com/owlim/RDFRank#</a> INSERT DATA ( rank:epsilon rank:setParam &quot;0.05&quot; . )</td>
</tr>
</tbody>
</table>

5.13.3.3 Full computation

To trigger the computation of the RDF Rank values for all resources, use the following update:

```
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 which means that the plugin manager will not be blocked during it 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 the Workbench, go to Setup → RDF Rank and click Compute Full.

```
RDF Rank
RDFRank for repository my_repo is with status [ RDFRank not built yet ] 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**.
5.13.3.4 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:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA {_:b1 rank:computeIncremental "true"}
```

computes RDF Rank values for the resources that do not have an associated value, i.e., the ones that have been added to the repository since the last full RDF Rank computation.

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 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.

5.13.3.5 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.

5.13.3.6 Checking the RDF Rank status

The RDF Rank plugin can be in one of the following statuses:

```java
/**
 * The ranks computation has been canceled
 */
CANCELED,

/**
 * The ranks are computed and up-to-date
 */
COMPUTED,

/**
 * A computing task is currently in progress
 */
COMPUTING,
```

(continues on next page)
You can get the current status of the plugin by running the following query:

```sparql
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?o WHERE { ?s rank:status ?o }
```

### 5.13.3.7 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:

```sparql
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 having a particular predicate or being in a particular named graph, you should include the predicate / graph IRI in one of the following lists: `includedPredicates` / `includedGraphs`. Empty lists are treated as wildcards. See below how to control the lists with SPARQL queries:

Get the content of a list:

```sparql
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?s WHERE { ?s rank:includedPredicates ?o }
```

Add an IRI to a list:

```sparql
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { <http:predicate> rank:includedPredicates "add" }
```

Remove an IRI from a list:

```sparql
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 ones 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/exclude all explicit/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, i.e., do not take part in the filtering. However, if you set them to false, they start acting as exclusion rules - this means they take precedence over the inclusion lists.

You can get the status of these parameters using:

```sql
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:includeExplicit _:b2 . }
```

You can set value of the parameters with:

```sql
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { rank:includeExplicit rank:setParam true }
```

### 5.13.4 Semantic similarity searches

#### 5.13.4.1 Why do I need the similarity plugin?

The similarity plugin allows exploring and searching semantic similarity in RDF resources.

As a user, you may want to solve cases where statistical semantics queries will be highly valuable, for example:

For this text (encoded as a literal in the database), return the closest texts based on a vector space model.

Another type of use case is the clustering of news (from a news feed) into groups by discussing events.

#### 5.13.4.2 What the similarity plugin does?

Humans determine the similarity between texts based on the similarity of the composing words and their abstract meaning. Documents containing by similar words are semantically related, and words frequently co-occurring are also considered close. The 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 - returns the closest semantically related terms
- Term to document - returns the most representative documents for a specific searched term
- Document to term - returns the most representative terms for a specific document
- Document to document - returns the closest related texts

#### 5.13.4.3 How the similarity plugin works?

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 to represent them into a vector space model representing 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.
Search similar terms

The example shows terms similar to “novichok” in the search index allNews. The term “novichok” is used in the search field. The selected option for both Search type and Result type is Term. Sample results of terms similar to “novichok”, listed by their score, are given below.

<table>
<thead>
<tr>
<th>documentID</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>novichok</td>
<td>1.00</td>
</tr>
<tr>
<td>porton</td>
<td>0.579221361785574</td>
</tr>
<tr>
<td>salisbury</td>
<td>0.539306692049809</td>
</tr>
<tr>
<td>nerve</td>
<td>0.537825357626078</td>
</tr>
<tr>
<td>skripal</td>
<td>0.480695938822142</td>
</tr>
<tr>
<td>skripals</td>
<td>0.46424357744188954</td>
</tr>
<tr>
<td>mirzayanov</td>
<td>0.4462164390828783</td>
</tr>
<tr>
<td>yulia</td>
<td>0.4394177895811721</td>
</tr>
<tr>
<td>poisoning</td>
<td>0.4359686538181075</td>
</tr>
<tr>
<td>agent</td>
<td>0.43257349963437026</td>
</tr>
</tbody>
</table>

Search documents for which selected term is specific

The term “novichok” is used as an example again. The selected option for Search type is Term, and for Result type is Document. Sample results of the most representative documents for a specific searched term, listed by their score, are given below.
The result with the highest score from the previous search is used in the new search. The selected option for Search type is Document, and for Result type is Term. Sample results of the most representative terms, listed by their score, are given below.
Search for closest documents

A search for the texts closest to the selected document is also possible. The same document is used in the search field. Sample results of the documents with the closest texts to the selected document - listed by their score - are given below. The titles of the documents prove that their content is similar, even though the sources are different.
5.13.4.4 Download data

To obtain the sample results listed above, you need to download data and create an index. Data from factforge.net is used in the following example. News from January to April 2018, together with their content, creationDate, and mentionsEntity triples, are downloaded.

Go to the SPARQL editor at [http://factforge.net/sparql](http://factforge.net/sparql) and insert the following query:

```
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 pubo:creationDate ?date .
}
```
WHERE {
  ?document pubo:creationDate ?date .
  FILTER (?p NOT IN (pubo:containsMention, pubo:hasFeature, pubo:hasImage))
  FILTER ( ?date > "2018-01-01"^^xsd:dateTime && ?date < "2018-04-30"^^xsd:dateTime )
}

Download the data using the Download As button, choosing the Turtle option. It will take some time to export the data to the query-result.ttl file.

Go to your local GraphDB instance and create a new repository called “news”.

Move the downloaded file to the <HOME>/graphdb-import folder so that it is visible in Import Server files (see here how to import server files).

Import the query-result.ttl file in your “news” repository.

Go to Setup and enable the Autocomplete index, and create an index for the “news” repository.

The Autocomplete index is used for autocompletion of URLs in the SPARQL editor and the View resource page.
5.13.4.5 Text based similarity searches

Create text similarity index

Create an index in the following way:

Go to Explore –> Similarity –> Create similarity index and change the search query to:

```sparql
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?documentID ?documentText {
  FILTER(isLiteral(?documentText))}
```

Please note that there are default parameters: `-termweight idf`

This will index the `allNews` content, where the ID of a document is the news piece’s IRI, and the text is the content.

Name the index `allNews`, save it and wait until it is ready.

Once the index has been created, you can see the following options on the right:

- With the `{…}` button, you can review or copy the SPARQL query for this index;
- The `Edit` icon allows you to modify the search query without having to build an index;
- You can also create a new index from an existing one;
- Rebuild the index;
- As well as delete it.

### Similarity indexes

![Existing Indexes](image)

Select one of your existing indexes to search in it

- `allNews`

### Create index parameters

- `-vectortype`: Real, Complex, and Binary Semantic Vectors
- `-dimension`: Dimension of semantic vector space, default value 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, then the orthogonality of the element vectors will be compromised leading 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, default value 10 except for when vectortype is binary, in which case default of dimension / 2 is enforced. For real and complex vectors default value is 10, but it is a good idea to use a higher value when the vector dimension is higher than 200. Simplest thing to do is to preserve this ratio, i.e., to divide the dimension by 20. It is worth mentioning that 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`, `sqrt`. It is a good idea to use term weighting when building indexes so we add `-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 has to occur in order to be indexed. Default value is set to 0, but it would be a bad idea to use it, as that would add a lot of big numbers/weird terms/misspelled words to the list of word vectors. 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 3-4, and start fine tuning from there.

• **maxfrequency**: Maximum number of times that a term can occur before getting removed from indexes. 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. 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`, index numbers or not.

• **mintermlength**: Minimum number of characters in a term.

• **indexfileformat**: Format used for serializing/deserializing vectors from disk, default lucene. Another option is text, may be used for debug to see the actual vectors. Too slow on real data.

### Disabled parameters

• **luceneindexpath**: Currently, you are not allowed to build your own Lucene index and create vectors from it since index + vectors creation is all done in one step.

• **stoplistfile**: Replaced by the [http://www.ontotext.com/graphdb/similarity/stopList](http://www.ontotext.com/graphdb/similarity/stopList) predicate. Stop words are passed as a string literal as opposed to a file.

• **elementalmethod**

• **docindexing**

### Stop words and Lucene Analyzer

In the *Stop words* field, 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.

In the *Analyzer class* field, set a Lucene analyzer to be used 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 as well.

Additionally, the Lucene connector also supports custom Analyzer implementations. This way you can create your own analyzer and add it to a classpath. The value of the *Analyzer Class* parameter must be a fully qualified name of a class that extends `org.apache.lucene.analysis.Analyzer`.

### Search in the index

Go to the list of indexes and click on *allNews*. For search options, select *Search type* to be either *Term* or *Document*. *Result type* can be either *Term* or *Document*. 

5.13. Plugins
Search parameters

- **-searchtype**: Different types of searches can be performed. Most involve processing combinations of vectors in different ways, in building a query expression, scoring candidates against these query expressions, or both. Default is sum that builds a query by adding together (weighted) vectors for each of the query terms, and search using cosine similarity. See more about SearchType here.

- **-matchcase**: If true, matching of query terms is case-sensitive; otherwise case-insensitive, default value is false.

- **-numsearchresults**: Number of search results.

- **-searchresultsminscore**: Search results with similarity scores below this threshold will not be returned, default value is -1.

See more about it at https://github.com/semanticvectors/semanticvectors/wiki/SearchOptions.

Delete or rebuild an index using a SPARQL query

To delete an index, use the following SPARQL query:

```sparql
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 "" .
}
```

To rebuild an index, simply create it again following the steps shown above.
Search in the index during rebuild with no downtime

GraphDB enables you to 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.

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 is introduced in order to reduce the searching times. Without a hashing algorithm, a search consists of the following steps:

1. A search vector is generated.
2. All vectors in 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. In order to speed up the process, hashing can be used to reduce the number of candidates for most similar vectors. This is where Locality-sensitive hashing can be very useful.

The Locality-sensitive hashing algorithm has two parameters that can be passed either during index creation, or as search option:

- `-lsh_hashes_num`: The number of random vectors used for hashing, default value is $\theta_0$.
- `-lsh_max_bits_diff`: The $m$ number of bits by which two hashes can differ and still be considered similar, default value is $\theta_0$.

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.

During a search, the workflow is as follows:

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 similar hashes like the one 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 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.
Depending on its configuration, the hash can perform in different ways.

A higher number of `-lsh_hashes_num` leads to more hash buckets with fewer elements in them. Conversely, a lower number of hashes would mean fewer but bigger buckets. The number of hashes leads to `2^n` potential buckets.

A higher number of `-lsh_max_bits_diff` leads to more buckets being checked, and vice versa. More precisely, an number of `-lsh_max_bits_diff` with an number of hashes leads to `m`-combinations of `n + (m - 1)` -combination of `n + ...` -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)`. However 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.

If you are looking to increase the 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 `-lsh_max_bits_diff` is too close to `-lsh_hashes_num`, the performance can be poorer compared to the default one because of the computational overhead.

Search similar news within days

The search can be extended by using the advantages of SPARQL to find the same news by different sources. This can be done by filtering all the news from the results related to a given period.

Click on View SPARQL Query, copy the query, and open the SPARQL editor to paste it there. Now you can integrate statistic similarity with RDF to obtain the following query:

```sparql
# Search similar news (SPARQL) within days

PREFIX similarity: <http://www.ontotext.com/graphdb/similarity/>
PREFIX similarity-index: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?entity ?score ?matchDate ?searchDate {
  ?search a similarity-index:allNews ;
  similarity:searchDocumentID ?searchDocumentID;
  similarity:searchParameters [ ];
  similarity:documentResult ?result .
  ?result similarity:value ?entity ;
  similarity:score ?score .
  ?entity pubo:creationDate ?matchDate .
  ?searchDocumentID pubo:creationDate ?searchDate .
```

(continues on next page)
Search for similar news, get their creationDate and filter only the news within the time period of two days.

Term to term search

The Term to term search gets the relevant terms by period.

Four separate indexes will be created as an example - for the news in January, February, March, and April.

Go to Create similarity index and create a new index with the following query:

```query
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/>

  ?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. For each month, go to the corresponding index and select the option for both Search type and Result type to be Term. Type “korea” in the search field. See how the results change with the time.
Notice how the score of “pyongyang” decreases with the end of the Olympics.

**Boosting a term’s weight**

It is possible to boost the weight of a given term 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 the results will be expected 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.
5.13.4.6 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. This way, the usual ways for constructing query vectors and searching for results in SemanticVectors can be used to suggest similar concepts based on the knowledge graph.

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.

Import the provided persons-1960-1970.ttl.

Create an Autocomplete index by enabling it from the Setup Autocomplete page.

For ease of use, you may add the following namespaces for the example dataset (done from Setup Namespaces):

- \texttt{dbo: http://dbpedia.org/ontology/}
- \texttt{dbr: http://dbpedia.org/resource/}
- \texttt{foaf: http://xmlns.com/foaf/0.1/}

Create predication-based index

Create a new predication-based similarity index from Explore Similarity Create similarity index. Select Create predication index.

Fill in the index name, and add the desired Semantic Vectors create index parameters. For example, it is a good idea to use term weighting when building indexes, so we will add \texttt{-termweight idf}. Also, for better results, set \texttt{-dimension} to higher than 200 which is the default.

Configure the Data query. This SPARQL SELECT query determines the data that will be indexed. The query must SELECT the following bindings:

- \texttt{?subject}
- \texttt{?predicate}
- \texttt{?object}
The Data query is executed during index creation to obtain the actual data for the index. When data in your repo changes, you need to also rebuild the index. 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 repo:

```
SELECT ?subject ?predicate ?object
WHERE {
}
```

Set the Search query. This SELECT query determines the data that will be fetched on search. The Search query is executed during search. Add more bindings by modifying this query to see more data in the results table.

For the first example, set the Search query to:

```
PREFIX similarity:<http://www.ontotext.com/graphdb/similarity/>
PREFIX similarity-index:<http://www.ontotext.com/graphdb/similarity/index/>
PREFIX psi:<http://www.ontotext.com/graphdb/similarity/psi/>
PREFIX rdfe: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

  ?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 . }
}
```

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. Additionally, if you want to edit an index query, you can do it for both the Search and the Analogical queries:
Search predication-based index

From the *Existing indexes*, select the index you want to search in.

In our example, we will be looking 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.

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”).

After importing the Nations.ttl sample dataset into a repository, build an Autocomplete index and a predication index following the steps in the documentation above.

Once your predication 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.

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 be greatly beneficial.
5.13.4.7 Hybrid indexing

When building a Predication index, it creates a random vector for each entity in the database, and uses these random vectors to generate the similarity vectors to be used later on 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 the previous parts of the page:

```
<express:donald-tusk-eu-poland-leave-european-union-polexit> <pubo:formattedDate> 1/11/2018
<telegraph:donald-tusk-warns-poland-could-hold-brexith-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.

In order to do so, you need to:

**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.

```
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)
}
```

Replacing `dateTime` with a simple string will enable you to create a Literal index.

At this stage, you should enable Autocomplete in case you have not enabled it yet, so as to make testing easier.

Go to Setup, and enable the Autocomplete index for allNews.

**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. This type of indexes can only be used as input indexes for predication indexes, and will be indicated in the Similarity page. They cannot 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:

```sql
SELECT ?documentID ?documentText {
  filter(isLiteral(?documentText))
  filter (?p !<http://factforge.net/ff2016-mapping/mentionsEntity>)
}
```

Use the Literal index

When creating the predication index from the “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:

```sql
SELECT ?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:

```sql
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 pubo: <http://ontology.ontotext.com/publishing#>

SELECT ?entity ?score ?content ?date {
  ?search a ?index ;
  ?searchType ?query;
  psi:searchPredicate ?psiPredicate;
  similarity:searchParameters ?parameters;

(continues on next page)
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.

Indexing behavior

When building the Literal index, it is 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, it would be expected 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 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 all literals, or at least all dates are indexed.

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 <literalIndex>` (replace `<literalIndex>` with the name of an existing Literal index): passed to a Predication index creates a hybrid index based on a Literal index

5.13.4.8 Training cycles

When building Text and Predication indexes, training cycles could be used 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 is time- and computationally consuming, and a higher number of cycles will greatly increase the building time.
5.13.5 JavaScript functions

In addition to internal functions, such as `NOW()`, `RAND()`, `UUID()`, and `STRUUID()`, GraphDB allows users to define and execute JavaScript code, further enhancing data manipulation with SPARQL. JavaScript functions are implemented within the special namespace `<http://www.ontotext.com/js#>`.

5.13.5.1 How to register a JS function

JS 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. It is possible to 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('');
    }
  }
}
```

Here is an example of how to retrieve a list of registered JS functions:

```prefix jsfn:<http://www.ontotext.com/js#>

PREFIX jsfn:<http://www.ontotext.com/js#>

SELECT ?s ?o {
  ?s jsfn:enum ?o
}
```

<table>
<thead>
<tr>
<th>s</th>
<th>o</th>
</tr>
</thead>
</table>
| isPalindrome     | function isPalindrome(str) {
|                  |   if (!str instanceof java.lang.String) return false;
|                  |   rev = reverse(str);
|                  |   return str.equals(rev);
|                  | }                    |
| reverse          | function reverse(str) {
|                  |   return str.split('').reverse().join('');
|                  | }                    |

`http://www.ontotext.com/js#enum` is a reserved predicate IRI for listing the available JS functions.

The following example registers a single function to return yesterday’s date:

```prefix jsfn:<http://www.ontotext.com/js#>

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)
We can then use this function in a regular SPARQL query, e.g., to retrieve 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 `xsd:date` and the output of the JS function, respectively.

### 5.13.5.2 How to remove a JS function

De-registering a JavaScript function is handled in the same fashion as registering one, with the only difference being 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              | [              | [              |
| isPalindrome   | function isPalindrome(str) { | function reverse(str) { |
| reverse        |     |     |
| getDateYesterday | UNDEFINED | 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.
5.13.6 Change tracking

GraphDB allows the tracking of changes that you have made in your data. Two tools offer this capability: the change tracking plugin, and the data history and versioning plugin.

5.13.6.1 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, e.g., 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 provides the ability to track the effects of SPARQL updates. These can be:

- Tracking what triples have been inserted or deleted;
- Distinguishing explicit from implicit triples;
- Running SPARQL using these triples.

5.13.6.2 Usage

The plugin introduces the following special graphs:

- http://www.ontotext.com/added/xxx – contains all added statements, including inferred ones
- http://www.ontotext.com/removed/xxx – contains 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:

   ```sql
   INSERT DATA {
   [1 <http://www.ontotext.com/track-changes> "xxx"
   }
   ```

   where xxx must be a unique ID assigned by the user.

3. Add or remove some data.
4. Commit the transaction.
5. Retrieve all added triples and their graph:

   ```sql
   SELECT * FROM <http://www.ontotext.com/added/xxx> {
   graph ?g {
   ?s ?p ?o
   }
   }
   ```

   where xxx is the ID assigned previously.

6. Retrieve the number of removed triples:

   ```sql
   SELECT (COUNT(*) as ?c) FROM <http://www.ontotext.com/removed/xxx> {
   ?s ?p ?o
   }
   ```

   where xxx is the previously assigned ID.

7. **CONSTRUCT** query using data that has just been added (advanced example):
8. Forget the tracked data:

```
INSERT DATA {
}
```

where \(xxx\) is the ID assigned previously.

**Note:** Note that 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()`.

### 5.13.7 Provenance

The provenance plugin enables the generation of 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, i.e., the ones inserted with a 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.

#### 5.13.7.1 Predicates

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

```
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
}
```

where \(xxx\) is the previously assigned ID.
5.13.7.2 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 ...
```

5.13.7.3 Usage and examples

1. In the Workbench SPARQL editor, add the following data as 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. Add the following data as 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#>

INSERT data {
  GRAPH ex:g1{
    ex:webcam_g1 a ex:OutdoorWebCam .
    ex:webcam_g1 ex:containedIn ex:skiresort .
  }
  GRAPH ex:g1a{
```

(continues on next page)
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:

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

The result will have two solutions.

4. This 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, e.g., `pr:derive (ex:g1 ex:g2)`.

```sparql
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 .
 }
}
```

This will return one solution since only the data from `ex:g1`, `ex:g2`, and `ex:g3` will be used, so that 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.
5.13.8 Proof plugin

5.13.8.1 What the plugin does

The GraphDB proof plugin enables you to find out how a particular statement has been derived by the inferencer, e.g., which rule fired and which premises have been matched to produce that statement.

The plugin is available as open source.

5.13.8.2 Predicates and namespace

The plugin supports the following predicates:

- **proof:explain** - the subject will be bound to the state variable (a unique BNode 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 (i.e., 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.

5.13.8.3 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 the 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 inverse property of urn:hasChild, and 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/>

  VALUES ( ?subject ?predicate ?object ) { ( <urn:Mary> <urn:hasChild> <urn:John> ) }
  ?ctx proof:subject ?s .
}
```

The result we get is:

<table>
<thead>
<tr>
<th>rule</th>
<th>s</th>
<th>p</th>
<th>o</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>urn:invOf</td>
<td>urn:childOf</td>
<td>urn:hasChild</td>
<td><a href="http://www.ontotext.com/implicit">http://www.ontotext.com/implicit</a></td>
</tr>
<tr>
<td>2</td>
<td>urn:invOf</td>
<td>urn:John</td>
<td>urn:childOf</td>
<td>urn:Mary</td>
</tr>
</tbody>
</table>

If we change the VALUES to:

```
VALUES ( ?subject ?predicate ?object ) {

  (<urn:John> <urn:childOf> <urn:Mary>)
}
```

we are getting:

<table>
<thead>
<tr>
<th>rule</th>
<th>s</th>
<th>p</th>
<th>o</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>explicit</td>
<td>urn:John</td>
<td>urn:childOf</td>
<td>urn:Mary</td>
</tr>
</tbody>
</table>

If we change the VALUES further to:

```
VALUES ( ?subject ?predicate ?object ) {

  (<urn:hasChild> owl:inverseOf <urn:childOf>)
}
```

308 Chapter 5. Usage
the result we get is:

<table>
<thead>
<tr>
<th>rule</th>
<th>s</th>
<th>o</th>
<th>p</th>
<th>a</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 &quot;owl:inverseOf&quot; urn:ChId owl:inverseOf urn:hasChild <a href="http://www.ontotext.com/implicit">http://www.ontotext.com/implicit</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Where we will get:

<table>
<thead>
<tr>
<th>rule</th>
<th>s</th>
<th>o</th>
<th>p</th>
<th>a</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &quot;owl:inverseOfBySymProp&quot; owl:inverseOf rdf:Type owl:SymmetricProperty <a href="http://www.ontotext.com/implicit">http://www.ontotext.com/implicit</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 owl1: <http://www.w3.org/2002/07/owl1#>

(continues on next page)
The results look as follows:

<table>
<thead>
<tr>
<th>stmt</th>
<th>0</th>
<th>rule</th>
<th>0</th>
<th>s</th>
<th>0</th>
<th>p</th>
<th>0</th>
<th>o</th>
<th>0</th>
<th>context</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Mary(rd1, type, Resource)&quot;</td>
<td>&quot;rule_rdf1_rdfs4a_4b_2&quot;</td>
<td>urn:John</td>
<td>urn:child</td>
<td>urn:Mary</td>
<td>urn:family</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;Mary(rd1,Child,John)&quot;</td>
<td>&quot;rule_rdf1_rdfs4a_4b_2&quot;</td>
<td>urn:child</td>
<td>urn:inverse</td>
<td>urn:hasChild</td>
<td>urn:netId</td>
<td><a href="http://www.ontotext.com/explicit">http://www.ontotext.com/explicit</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&quot;Mary(rd1,Child,John)&quot;</td>
<td>&quot;rule_rdf1_rdfs4a_4b_2&quot;</td>
<td>urn:John</td>
<td>urn:child</td>
<td>urn:Mary</td>
<td>urn:family</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```
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>.  
  <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>;  
```
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` (`MerloGrape`, `CabernetGrape` etc.) and `Malt` (`PilsenerMalt`, `WheatMalt` etc.);
- classes `RedWine` and `WhiteWine` are declared as intersections of `Wine` with `RedThings` or `WhiteThings` with `WhiteWine`, respectively;
- finally, we introduce 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 create a `stmt()` 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) {
    return `('+lname(s)+', '+lname(p)+', '+lname(o)+c?' , '+lname(c):''+');
  }
  ...
}
```

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

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

Now, let’s see how the (urn:Merlo rdf:type urn:RedWine) has been derived (note the use of 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.

```sparql
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:subject ?s.
}
```

The result looks like this:

<table>
<thead>
<tr>
<th>stmt</th>
<th>rule</th>
<th>premise</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(Merlo, type, RedWine)]</td>
<td>&quot;rule_owl_typeByIntersect_1&quot;</td>
<td>&quot;RedWine, intersectionOf (_node1, explicit)&quot;</td>
</tr>
<tr>
<td>[(Merlo, type, RedWine)]</td>
<td>&quot;rule_owl_typeByIntersect_1&quot;</td>
<td>&quot;Merlo, allTypes, _node2, explicit&quot;</td>
</tr>
</tbody>
</table>

The first premise is explicit, and comes from the definition of 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:

```sparql
Id: owl_typeByIntersect_1

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

where a is bound to Merlo and c to RedWine.

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:

<table>
<thead>
<tr>
<th>stmt</th>
<th>a</th>
<th>rule</th>
<th>premise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>rule_owl_typeByIntersect_3</td>
<td></td>
</tr>
</tbody>
</table>

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

```
Id: owl_typeByIntersect_3

a <rdf:first> b
da <rdf:type> b
d <rdf:rest> c
<onto:_allTypes> c
--------------------------------------------------------------------------
da <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:
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_typeByIntersection_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:
The statement \((Merlo, \_\text{allTypes}, \_\text{node2})\) was derived by \texttt{owl\_typeByIntersect\_2} and the only implicit statement matching as premise is \((Merlo, \text{type}, \text{Wine})\).

The \texttt{owl\_typeByIntersect\_2} rule looks like this:

```java
Id: owl_typeByIntersect_2

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

where \(c\) is bound to \textit{Merlo} and \(b\) to \textit{Wine}.

Let’s add the \((\textit{Merlo, type, Wine})\) to the \texttt{SUBSELECT} we used to explore as another \texttt{UNION} using \texttt{VALUES}, and evaluate the query again:

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

The results have now been expanded, as shown in lines 13-16:
These come from rule owl_typeBySomeVal where all premises matching it were explicit. The rule looks like:

```
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, etc.

In conclusion, while the chain is rather obscure, we were able to find out how the inferencer derived:

- (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>).

### 5.13.9 Graph path search

#### 5.13.9.1 Overview

The GraphDB Graph path search functionality 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, or 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.
It 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 amount 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.

**Hint:** Graph path search is similar to the SPARQL 1.1 *property paths* feature as both enable graph traversal, allowing you to 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 it and load it into a GraphDB repository via Import User data Upload RDF files.

### 5.13.9.2 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:

- All of the shortest paths with the shortest length are returned. 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 and/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 using a literal as a destination. Both source and destination can be literals (e.g., 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
<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path:shortestPath</td>
<td><strong>Required</strong> 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.</td>
</tr>
<tr>
<td>path:allPaths</td>
<td><strong>Required</strong> property that finds all paths between two nodes or between all nodes and the starting node.</td>
</tr>
<tr>
<td>path:distance</td>
<td><strong>Required</strong> property that finds the distance of the shortest path between two resources, which is the number of edges that connect the resources.</td>
</tr>
<tr>
<td>path:cycle</td>
<td><strong>Required</strong> property that finds cyclic dependencies for a given resource, meaning that a resource points back to itself.</td>
</tr>
</tbody>
</table>

**Modifier Bindings**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>Supported algorithms</th>
</tr>
</thead>
</table>
| path:poolSize      | **Optional** modifier that enables parallel path search query evaluation. The parameter allows you to 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, i.e., the more licensed cores, the larger the pool size and the faster queries. See also *Parallel search mode*. | Shortest path  
All paths  
Shortest distance  
Cyclic path |

**Variable Bindings**
<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>Supported algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>path:sourceNode</td>
<td>Required variable binding that specifies the source node from which the path search commences. If a destination is selected, this variable can be optional.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortest distance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:destinationNode</td>
<td>Required variable binding that specifies the destination node where the path traversal completes. If a source is selected, this variable can be optional.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortest distance</td>
</tr>
<tr>
<td>path:distanceBinding</td>
<td>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.</td>
<td>Shortest distance</td>
</tr>
<tr>
<td>path:resultBinding</td>
<td>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.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:startNode</td>
<td>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 will return the interim source node for each edge inside the path. Equally good to use with and without graph pattern in cases where we do not care about the entire edge.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortest distance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:endNode</td>
<td>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 will return the interim destination node for each edge inside the path. Equally good to use with and without pattern in cases where we do not care about the entire edge.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortest distance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:propertyBinding</td>
<td>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.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:resultBindingIndex</td>
<td>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.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
</tbody>
</table>
Filtering Parameters

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>Supported algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>path:minPathLength</td>
<td>Optional variable binding that specifies the minimal path length returned by an all paths search. This property is inclusive (meaning that a min 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.</td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
<tr>
<td>path:maxPathLength</td>
<td>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 max 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.</td>
<td>Shortest path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortest distance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyclic path</td>
</tr>
</tbody>
</table>

Required properties include a binding for source and/or destination, as well as the type of the search.

Optional properties include min/max 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.

Search algorithms

Shortest path

The algorithm finds the shortest path between two input nodes 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.

The shortest path algorithm can be used with a wildcard predicate as well as a graph pattern that is used as an edge abstraction. With it, we can impose filtering through property negation or selection, define indirect relationships, specify named graphs, etc.
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 are required to be present at least once inside the graph pattern.

See examples of how Shortest path search is used here.

**All paths**

This algorithm finds all paths between two nodes or between all nodes and the starting/destination node. It can be used with a wildcard predicate, as well as with more complex graph patterns and relationships. With it, we can also impose filtering with min/max number of edges, and can include or exclude inferred edges.

See examples of how All paths search is used here.

**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. The nodes themselves will not be returned as results, only the distance.

See an example of how Shortest distant search is used here.

**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 min/max values.

See an example of how Cyclic path search is used here.

**Search modifiers**

**Parallel search mode**

This mode enables parallel path search query evaluation and allows you to specify the size of the 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, i.e., the more licensed cores, the larger pool size and faster 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.

See an example of how Parallel path search is used here.

**Exportable graph pattern bindings**

Export bindings allow you to project any number of bindings from the graph pattern query service. The power of SPARQL graph pattern-matching property paths is combined with GraphDB’s path search algorithm, enabling the user to restrict the start and the end nodes of the path search to those pairs that match a particular graph pattern defined as 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 have to 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 have to be present in the nested graph pattern service.

See an example of how Export bindings are used here.

**Bidirectional search**

The bidirectional search functionality can be used to traverse paths as if the graph is undirected, i.e., as if 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. It can be used 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 value true of type xsd:boolean.

See an example of how Bidirectional graph path search is used here.

### 5.13.9.3 Usage examples

#### 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 wildcard predicate**

Suppose we want to find the shortest path between source node - the movie “The Black Panther (1977)”, and destination node - Marvel Comics’ creative leader Stan Lee.

In the Workbench SPARQL editor, run the following query:

```sparql
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 by using 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.

We can also trace the path in the **Visual graph** of the Workbench.

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 preferred predicates. (In our case here: **differentFrom, studio, founder, and keyPerson**)

---

**Visual graph**

![Visual graph diagram](image-url)
Shortest path search with graph pattern

In this query, we will again be searching 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.

```sparql
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, i.e., 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:


<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>index</th>
<th>path</th>
</tr>
</thead>
</table>

In the Visual graph, it will look like this:
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 ;
    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:
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 functionality and in most cases would not display every path returned by the path search query.

All paths search with unbound destination

Now, let’s find 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 is more than one way 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:
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. 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 dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
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 db: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/resource/>
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:


```
Note: Keep in mind that when using graph patterns, we cannot view the predicates connecting the nodes. Thus, when exploring the path edges as RDF-star statements, the predicate http://www.ontotext.com/path#connectedTo is generated.
```
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
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 .
  }
}
```

The first two returned paths are:
Shortest distance

The next query finds the shortest distance between source node Marvel Studios and a date literal which represents Marvel Studios President Kevin Feige’s birthday.

```sparql
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.

### Filter query results

<table>
<thead>
<tr>
<th>edge</th>
<th>0</th>
<th>index</th>
<th>0</th>
<th>path</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://dbpedia.org/resource/Chris_Evans_(actor)">http://dbpedia.org/resource/Chris_Evans_(actor)</a> path:connectedTo <a href="http://dbpedia.org/resource/Mark_Ruffalo">http://dbpedia.org/resource/Mark_Ruffalo</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Mark_Ruffalo">http://dbpedia.org/resource/Mark_Ruffalo</a> path:connectedTo <a href="http://dbpedia.org/resource/Chris_Hemsworth">http://dbpedia.org/resource/Chris_Hemsworth</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Chris_Evans_(actor)">http://dbpedia.org/resource/Chris_Evans_(actor)</a> path:connectedTo <a href="http://dbpedia.org/resource/Cobie_Smulders">http://dbpedia.org/resource/Cobie_Smulders</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Cobie_Smulders">http://dbpedia.org/resource/Cobie_Smulders</a> path:connectedTo <a href="http://dbpedia.org/resource/Mark_Ruffalo">http://dbpedia.org/resource/Mark_Ruffalo</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Mark_Ruffalo">http://dbpedia.org/resource/Mark_Ruffalo</a> path:connectedTo <a href="http://dbpedia.org/resource/Chris_Hemsworth">http://dbpedia.org/resource/Chris_Hemsworth</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can see that the shortest path connecting them consists of two edges.
Cyclic path

The following query finds all paths that begin and end with source node Marvel Studios.

```sparql
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:cycles .
    path:sourceNode ?src ;
    path:resultBinding ?edge ;
    path:pathIndex ?path ;
    path:resultBindingIndex ?index .
  }
}
```

The first three returned paths will be:

<table>
<thead>
<tr>
<th>edge</th>
<th>index</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://dbpedia.org/resource/Marvel_Studios">http://dbpedia.org/resource/Marvel_Studios</a></td>
<td>0</td>
<td>?<strong>val</strong></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Marvel_Studios">http://dbpedia.org/resource/Marvel_Studios</a></td>
<td>0</td>
<td>?<strong>val</strong></td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Marvel_Studios">http://dbpedia.org/resource/Marvel_Studios</a></td>
<td>0</td>
<td>?<strong>val</strong></td>
</tr>
</tbody>
</table>

In the visual graph:
Parallel search mode

To demonstrate this functionality, let’s use the Shortest path search with wildcard predicate example from earlier. To perform parallel path search, you need to set the path:poolSize property:

```sparql
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:poolSize 8 ;
    path:resultBindingIndex ?edgeIndex ;
    path:resultBinding ?edge ;
  }
}
```

The query will return the same results but execute faster.
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.

```sql
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/>

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 .
  }
}
```

The first six returned paths will be:
Which in the visual graph would look like this:

**Visual graph**

Bidirectional search

This query finds the shortest bidirectional path between source node **The Black Panther** movie from 1977 and destination node **Marvel Studios**.

```sparql
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) {
    ("http://dbpedia.org/resource/Chris_Evans" "http://dbpedia.org/resource/Mark_Ruffalo")
  } WHERE {
    ?src path:hasEdge ?edge.
    ?edge path:hasPath ?path.
    ?path path:hasLabel "Avengers: Age of Ultron".
    FILTER (strstarts(str(?path), "http://dbpedia.org/resource/"))
  }
}
```

(continues on next page)
The first two returned bidirectional paths are:

<table>
<thead>
<tr>
<th>edge</th>
<th>index</th>
<th>path</th>
</tr>
</thead>
</table>

In the visual graph:
5.13.10  Autocomplete index

5.13.10.1 What the index does

The Autocomplete index offers suggestions for the IRIs’ local 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.

In case you are getting peculiar results and you think the index might be broken, use the Build Now button.

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.

5.13.10.2 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 config as label predicate, then the text of the ?label is tokenized and indexed for this IRI. You can add a new label via the right-hand button in the Autocomplete screen, which will open this dialog box:
Local name tokenization

Local names are split by special characters (e.g., _, -), or in cases when they contain camelCase and/or numbers. For example:

<table>
<thead>
<tr>
<th>IRI</th>
<th>Local name tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://dbpedia.org/resource/Bulgarian_Tournament_Cup">http://dbpedia.org/resource/Bulgarian_Tournament_Cup</a></td>
<td>Bulgarian Tournament Cup</td>
</tr>
<tr>
<td><a href="http://dbpedia.org/resource/Post-rock">http://dbpedia.org/resource/Post-rock</a></td>
<td>Post rock</td>
</tr>
<tr>
<td><a href="http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#ChardonnayGrape">http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#ChardonnayGrape</a></td>
<td>Chardonnay Grape</td>
</tr>
<tr>
<td><a href="http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#USRegion">http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#USRegion</a></td>
<td>US Region</td>
</tr>
</tbody>
</table>

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 `wineWhite`.

Some examples:

<table>
<thead>
<tr>
<th>Search string</th>
<th>Found IRI</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Tour”</td>
<td><a href="http://dbpedia.org/resource/Bulgarian_Tournament_Cup">http://dbpedia.org/resource/Bulgarian_Tournament_Cup</a></td>
</tr>
<tr>
<td>“white win” OR “whiteWin”</td>
<td><a href="http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WhiteWine">http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WhiteWine</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WhiteTableWine">http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WhiteTableWine</a></td>
</tr>
</tbody>
</table>
5.13.10.3 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 / Cmd+Space** depending on your OS and the way you have set up your shortcuts.

You can use autocompletion to:

- Search for a single word in all IRIs:

  ![Autocomplete example 1](image1)

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

  ![Autocomplete example 2](image2)

- Search for more than one word:

  ![Autocomplete example 3](image3)

- Indexed text is split where digits or digit sequences are found, so you can also search by number:

  ![Autocomplete example 4](image4)
5.13.10.4 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.

5.13.10.5 Workbench queries

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 {
}
```

- 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"
}
```

5.13.11 GeoSPARQL support

5.13.11.1 What is GeoSPARQL

GeoSPARQL is a standard for representing and querying geospatial linked data for the Semantic Web 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 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 Open Geospatial Consortium (OGC). You can input data of all known CRS types - it will be properly indexed by the plugin, and you will also be able to query it in both the default CRS84 format and in the format in which it was imported.

The following is a simplified diagram of the GeoSPARQL classes Feature and Geometry, as well as some of their properties:

5.13.11.2 Usage

Configuration parameters

The following parameters can be used when configuring the plugin:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Predicate</th>
<th>Description</th>
<th>Default</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td><a href="http://www.ontotext.com/plugins/geosparql#enabled">http://www.ontotext.com/plugins/geosparql#enabled</a></td>
<td>Enables and disables plugin</td>
<td>false</td>
<td>PREFIX geoSparql: <a href="http://www.ontotext.com/plugins/geosparql#">http://www.ontotext.com/plugins/geosparql#</a> INSERT DATA { [] geoSparql:enabled &quot;true&quot; . }</td>
</tr>
<tr>
<td>prefixTree</td>
<td><a href="http://www.ontotext.com/plugins/geosparql#prefixTree">http://www.ontotext.com/plugins/geosparql#prefixTree</a></td>
<td>Implementation of the tree used while building the index; stores value before rebuilding</td>
<td>prefixTree.QUAD</td>
<td>PREFIX geoSparql: <a href="http://www.ontotext.com/plugins/geosparql#">http://www.ontotext.com/plugins/geosparql#</a> INSERT DATA { [] geoSparql:prefixTree &quot;geohash&quot; . }</td>
</tr>
<tr>
<td>Parameter</td>
<td>precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#precision&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Specifies the desired precision; stores value before rebuilding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>11 min value 1; max value depends on used prefixTree or (24 for geohash and 50 for QUAD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:precision &quot;11&quot; . }</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>currentPrefixTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#currentPrefixTree&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>Value of last built index</td>
</tr>
<tr>
<td>Default</td>
<td>PrefixTree.QUAD</td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:currentPrefixTree &quot;geohash&quot; . }</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>currentPrecision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#currentPrecision&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>Value of last built index</td>
</tr>
<tr>
<td>Default</td>
<td>11</td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:currentPrecision &quot;11&quot; . }</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>maxBufferedDocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#maxBufferedDocs&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>Speeds up building and rebuilding of index</td>
</tr>
<tr>
<td>Default</td>
<td>1,000 (max. allowed 5,000)</td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:maxBufferedDocs &quot;3000&quot; . }</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ramBufferSizeMB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#ramBufferSizeMB&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>Speeds up building and rebuilding of index</td>
</tr>
<tr>
<td>Default</td>
<td>32.0 (max. allowed 512.0)</td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:ramBufferSizeMB &quot;256.0&quot; . }</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>ignoreErrors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td><code>&lt;http://www.ontotext.com/plugins/geosparql#ignoreErrors&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>Ensures building of the index even in case of erroneous data</td>
</tr>
<tr>
<td>Default</td>
<td>false</td>
</tr>
<tr>
<td>Example</td>
<td><code>PREFIX geoSparql: &lt;http://www.ontotext.com/plugins/geosparql#&gt; INSERT DATA { [] geoSparql:ignoreErrors &quot;true&quot; . }</code></td>
</tr>
</tbody>
</table>
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.

```sparql
PREFIX geosparql: <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA { [] geosparql:enabled "true" . }
```

**Note:** All functions require as input WKT or GML literals while the predicates expect resources of type `geo:Feature` or `geo:Geometry`. The GraphDB implementation has a non-standard extension that allows you to use literals with the predicates too. See [Example 2 (using predicates)] for an example of that usage.

**Warning:** All GeoSPARQL functions starting with `geof:` like `geof:sfOverlaps` do not use any indexes and are always enabled! That is why it is recommended to use the indexed operations like `geo:sfOverlaps`, whenever it is possible.

Disable plugin

When the plugin is disabled, it does not index any data or process updates. It does not handle any of the GeoSPARQL predicates either.

```sparql
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`. To check the current runtime configuration:

```sparql
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 11 precision value of 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.

After changing the indexing algorithm, you need to trigger a reindex.

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: However, do not set these values too high, otherwise 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.

Ignore errors on indexing

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 in the log. 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 `ext:` stands for the namespace `http://rdf.usekm.com/ext#`. The types `geo:Geometry`, `geo:Point`, etc. refer to GeoSPARQL types in the `http://www.opengis.net/ont/geosparql#` namespace.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:double ext:area(geomLiteral g)</code></td>
<td>Calculates the area of the surface of the geometry.</td>
</tr>
<tr>
<td><code>geomLiteral ext:closestPoint(geomLiteral g1, geomLiteral g2)</code></td>
<td>For two given geometries, computes the point on the first geometry that is closest to the second geometry.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:containsProperly(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry properly contains the second geometry. <code>Geom1</code> contains properly <code>geom2</code> if all <code>geom1</code> contains <code>geom2</code> and the boundaries of the two geometries do not intersect.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:coveredBy(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry is covered by the second geometry. <code>Geom1</code> is covered by <code>geom2</code> if every point of <code>geom1</code> is a point of <code>geom2</code>.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:covers(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry covers the second geometry. <code>Geom1</code> covers <code>geom2</code> if every point of <code>geom2</code> is a point of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>xsd:double ext:hausdorffDistance(geomLiteral g1, geomLiteral g2)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>geo:Line ext:shortestLine(geomLiteral g1, geomLiteral g2)</code></td>
<td>Computes the shortest line between two geometries. Returns it as a LineString object.</td>
</tr>
<tr>
<td><code>geomLiteral ext:simplify(geomLiteral g, double d)</code></td>
<td>Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm.</td>
</tr>
<tr>
<td><code>geomLiteral ext:simplifyPreserveTopology(geomLiteral g, double d)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:isValid(geomLiteral g)</code></td>
<td>Checks whether the input geometry is a valid geometry.</td>
</tr>
</tbody>
</table>

5.13.11.3 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

```sparql
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

```sparql
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 .
  FILTER (!sameTerm(?aGeom, ?fGeom))
}
```
Example 1 result

<table>
<thead>
<tr>
<th>?f</th>
</tr>
</thead>
<tbody>
<tr>
<td>my:B</td>
</tr>
<tr>
<td>my:F</td>
</tr>
</tbody>
</table>

Example 2

Find all features that are within a transient bounding box geometry, where spatial calculations are based on my:hasPointGeometry.

Using a function

```sparql
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.

```sparql
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

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(geo: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.openlinksw.com/g/geo>
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.openlinksw.com/g/geo>
PREFIX my: <http://example.org/ApplicationSchema#>

SELECT ?f WHERE {
  ?f geo:sfOverlaps my:AExactGeom
}
```
Example 5 result

```sparql
?f
my:D
my:DExactGeom
```

**Note:** The example in the GeoSPARQL specification has additional results `my:E` and `my:EEexactGeom`. In fact, `my:E` and `my:EEexactGeom` do not overlap `my:AexactGeom` because they are of different dimensions (`my:AexactGeom` is a Polygon and `my:EEexactGeom` is a LineString) and the `overlaps` relation is defined only for objects of the same dimension.

**Tip:** For more information on GeoSPARQL predicates and functions, see the current official spec: OGC 11-052r4, Version: 1.0, Approval Date: 2012-04-27, Publication Date: 2012-09-10.

### 5.13.12 Geospatial extensions

#### 5.13.12.1 What are geospatial extensions

GraphDB provides support for 2-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, i.e., 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:
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpatialThing</td>
<td>A class for representing anything with a spatial extent, i.e., size, shape, or position.</td>
</tr>
<tr>
<td>Point</td>
<td>A class for representing a point (relative to Earth) defined by latitude, longitude (and altitude). subClassOf <a href="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing">http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</a></td>
</tr>
<tr>
<td>location</td>
<td>The relation between a thing and where it is. Range SpatialThing subPropertyOf <a href="http://xmlns.com/foaf/0.1/based_near">http://xmlns.com/foaf/0.1/based_near</a></td>
</tr>
<tr>
<td>lat</td>
<td>The WGS84 latitude of a SpatialThing (decimal degrees). domain <a href="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing">http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</a></td>
</tr>
<tr>
<td>long</td>
<td>The WGS84 longitude of a SpatialThing (decimal degrees). domain <a href="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing">http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</a></td>
</tr>
<tr>
<td>lat_long</td>
<td>A comma-separated representation of a latitude, longitude coordinate.</td>
</tr>
<tr>
<td>alt</td>
<td>The WGS84 altitude of a SpatialThing (decimal meters above the local reference ellipsoid). domain <a href="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing">http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</a></td>
</tr>
</tbody>
</table>

5.13.12.2 How to create a geospatial index

Execute the following INSERT query:

```
PREFIX ontgeo: <http://www.onotext.com/owlim/geo#>
INSERT DATA { _:b1 ontgeo: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, i.e., no statements describing resources with latitude and longitude properties, this update query will fail.

5.13.12.3 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.onotext.com/owlim/geo#>
<table>
<thead>
<tr>
<th>Construct</th>
<th>Nearby (lat long distance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>?point omgeo:nearby(?lat ?long ?distance)</code></td>
</tr>
</tbody>
</table>
| 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  
| 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 .`  
|           | `?link omgeo:nearby(?latBase ?longBase "50mi") .`  
|           | `}`
<table>
<thead>
<tr>
<th>Construct</th>
<th>Within (rectangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>?point omgeo:within(?lat1 ?long1 ?lat2 ?long2)</code></td>
</tr>
</tbody>
</table>
| Description | This statement pattern is used to test/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:  
  - `?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:  
  ```sparql  
  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-pos:lat ?lat .  
  }  
  ``` |
### Construct

**Within (polygon)**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>?point omgeo:within(?lat1 ?long1 ... ?latN ?longN)</th>
</tr>
</thead>
</table>

**Description**

This statement pattern is used to test/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:

- 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:

```sparql
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" "-3.86"
  "55.81" "-2.83"
  "52.74" "1.68"
  "51.17" "1.41" ) .
  ?link geo-pos:lat ?lat .
}
```

### 5.13.12.4 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

**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 caves in the sides of cliffs lying within a polygon approximating the shape of England:

```sparql
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 .
  ?a2 geo-ont:name "Brize Norton" .
  ?a2 geo-pos:lat ?lat3 .
  FILTER( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) < 80 )
}
ORDER BY ASC( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) )
```

---

5.13.12.5 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 2-dimensional points are supported**, i.e., there is no special handling of `geo:alt` (metres 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:nearby()` **may not work correctly** when the region (polygon or rectangle) spans the +/-180 meridian;
- `omgeo:nearby()` **uses the great circle distance** between points.
5.13.13 Data history and versioning

5.13.13.1 What the plugin does

The Data history and versioning plugin enables you to access past states of your database through versioning of the RDF data model level. Collecting and querying the history of a database is beneficial for users and organizations that want to preserve all of their historical data, and are often faced with the common use case: I want to know when a value in the database has changed, and what the previous system state in time was.

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.

It can be useful in several main types of 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;
- Answering the question of 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 of the before/after situation”;
- Maintaining history only for specific classes or properties, i.e., no need for keeping history for everything. This is 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.

5.13.13.2 Index components

The plugin index is of the type DSPOCI, meaning that it consists of the following components:

- **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 as minimum bits as it makes 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 ordered in descending order (most recent updates come first), and all other components are ordered in ascending order. For example:

<table>
<thead>
<tr>
<th>Date-time</th>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
<th>Context</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>1570623056397</td>
<td>34</td>
<td>1</td>
<td>29</td>
<td>-3</td>
<td>TRUE</td>
</tr>
<tr>
<td>1570623056397</td>
<td>34</td>
<td>1</td>
<td>38</td>
<td>-2</td>
<td>TRUE</td>
</tr>
<tr>
<td>1570623042812</td>
<td>34</td>
<td>1</td>
<td>30</td>
<td>-2</td>
<td>FALSE</td>
</tr>
<tr>
<td>1570623042812</td>
<td>34</td>
<td>2</td>
<td>31</td>
<td>-2</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
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 using predicate parameters as described in the examples below.

5.13.13.3 Usage

Enable/disable 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 query:

```
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
}
```

Clear all data

If you want to clear 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 only the history without deleting the data in the repository or having to disable collecting history. Execute:

```
PREFIX hist: <http://www.ontotext.com/at/>
INSERT DATA {
    [] hist:clearHistory [] .
}
```
Trim history

The history can also be trimmed in various ways:

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 interpretable as `xsd:date` or `xsd:dateTime`. If only the date is specified, the time is assumed to be midnight (00:00:00). The timezone is by default the system timezone. For more precise trimming, a full datetime 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 interpretable as `xsd:duration`. P3D here means 3 days - so only the history from the last 3 days would remain after executing the update. We can also specify minutes, hours, etc.

History filtering

As keeping history for everything is most of the time unnecessary, as well as quite time- and resource-consuming, this plugin provides the capability for specifying only certain classes or properties. 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` - the type of the entity on this position must be the specified one, case insensitive
- 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 have 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
}
```

5.13.13.4 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> .
}
```

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" };
insert data { <urn:Kirk> <urn:name> "James Tiberius Kirk" }
```

3. 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 * {
  ?log a hist:history ;
  hist:timestamp ?time ;
  hist:graph ?g ;
  hist:subject ?s ;
  hist:predicate ?p ;
  hist:object ?o ;
  hist:insert ?i
}
```

The retrieved results are in descending order, i.e., the most recent change comes first:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. You can also find out what changes were made for a subject and a predicate within a specific time period between moment A and moment B. This is done with the hist:parameters predicate used the following way: ?log hist:parameters (?fromDateTime ?toDateTime ?subject ?predicate ?object ?context).

While the predicate is not mandatory, passing parameters when querying history is much more efficient than fetching all history elements and then filtering them. Note 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 * {
  ?log a hist:history ;
  hist:parameters ('2022-07-12T17:17:00'^^xsd:dateTime , '2022-07-12T17:17:00'^^xsd:dateTime);
  hist:timestamp ?time ;
  hist:graph ?g ;
  hist:subject ?s ;
  hist:predicate ?p ;
  hist:object ?o ;
  hist:insert ?i
}
```

You can also find out all changes for a particular subject and predicate. Note that the ?fromDateTime ?toDateTime parameters are left unbound.
2. 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:

```sql
# Return data as it looked on 2022-07-12 16:17:17 server time
# ^20220712161717 { ?officer <urn:name> ?name ;
# <urn:rank> ?rank ;
# <urn:dateOfBirth> ?dateOfBirth .
}
```

The same query will return a valid graph with only the date specified:

```sql
# Return data as it looked on 2022-07-12 00:00:00 server time
# (explicit year and month only)
# <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:

```sql
DESCRIBE <urn:Kirk> from <http://www.ontotext.com/at/20220712161717>
```

The result from our example at that point in time would be:
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.

5.13.14 Text mining plugin

5.13.14.1 What the plugin does

The GraphDB text mining plugin allows you to consume the output of text mining APIs as SPARQL binding variables. Depending on the annotations returned by the concrete API, the plugin enables multiple use cases like:

- 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 in a combination with the Kafka GraphDB connector

The plugin readily supports the protocols of these services:

- *spaCy server*
- *GATE Cloud*
- *Ontotext’s Tag API*

In addition, any text mining service that provides response as JSON can be used when you provide a JSLT transformation to remodel the output from the service output to an output understandable by the plugin. See the below examples for querying the *Google Cloud Natural Language API* and the *Refinitiv API* using the generic client.

5.13.14.2 Usage examples

A typical use case would be having a piece of text (for example news content), in which we want to recognize people, organizations, and locations fragments. Ideally, we will link them to entity IRIs that are already known in the knowledge graph, i.e., Wikidata or PermID IRIs providing infinite 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.”

Let’s find out what annotations the different services will find in the text.
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.

Currently, the NER pipeline is the only spaCy component supported by the text mining plugin.

Create a spaCy client

1. Run the spaCy server through its Docker image with the following commands:

   ```
   sudo 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:

   ```sparql
   PREFIX ttxtm: <http://www.ontotext.com/textmining#>
   PREFIX ttxtm-inst: <http://www.ontotext.com/textmining/instance#>
   INSERT DATA {
     ttxtm-inst:localSpacy ttxtm:connect ttxtm:Spacy;
   }
   ```

   where http://localhost:8000 is the location of the spaCy server set up using the above Docker image.

Note that 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:

```sparql
PREFIX ttxtm: <http://www.ontotext.com/textmining#>
PREFIX ttxtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
  ttxtm-inst:localSpacy ttxtm:connect ttxtm:Spacy;
  ttxtm:service "http://localhost:8000";
  ttxtm:sense2vec "false" .
}
```

Find spaCy entities through GraphDB

The simplest query will return all annotations with their types and offsets. Since spaCy also provides sentence grouping, for each annotation, we can get the text it is found in.

```sparql
PREFIX ttxtm: <http://www.ontotext.com/textmining#>
PREFIX ttxtm-inst: <http://www.ontotext.com/textmining/instance#>
WHERE {
  ?searchDocument a ttxtm-inst:localSpacy;
  ttxtm:text "Dyson Ltd. plans to hire 450 people globally, with more than half of 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 ttxtm-inst:localSpacy {
    ?annotation ttxtm:annotationText ?annotationText ;
    ttxtm:annotationKey ?annotationKey;
}
```
WHERE

WHERE {
  ?searchDocument a txtm:instance: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:instance:localSpacy {
    ?annotation txtm:annotationText ?annotationText ;
    txtm:annotationType ?annotationType ;
    txtm:annotationStart ?annotationStart ;
    txtm:annotationEnd ?annotationEnd ;
    optional {
      }
    }
    }
  }
  }
  }
  }
  }
  }
  }
  }
  }
  }
  }
The sense2vec similarity feature provides us with the additional knowledge that Dyson is somehow related to “vacuums” and “Miele”.

GATE Cloud

GATE Cloud is a text analytics as a service that provides various pipelines. Its ANNIE named entity recognizer used by the plugin identifies basic entity types, such as Person, Location, Organization, Money amounts, Time and Date expressions.

Create 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;
}
```

Obviously, you can provide the annotation types you are interested in using the query parameters.
Find GATE entities through GraphDB

```sql
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
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'" .

  graph txtm-inst:gateService {
    ?annotation txtm:annotationText ?annotationText ;
    txtm:annotationType ?annotationType ;
    txtm:annotationStart ?annotationStart ;
    txtm:annotationEnd ?annotationEnd ;
  }
}
```

In GATE, sentences are returned as annotations, so they will appear as annotations in the response.

```
| 1 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "money" | "unknown"
| 2 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "Gately" |
| 3 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 4 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 5 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 6 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 7 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 8 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 9 | "Dyson Ltd"  | "Organization" | 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 10| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 11| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 12| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 13| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 14| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 15| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 16| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 17| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 18| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 19| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 20| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 21| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 22| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 23| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 24| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 25| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 26| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 27| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 28| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 29| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 30| "Dyson Ltd"  | "Organization"| 97 | "mention" | 97 | "mention" | "rule"   | "unknown" |
| 31| "Dyson Ltd plans to hire 450 people globally, with more than half the recruits in its headquarters in Singapore."| "Sentence"| 97 | "mention" | 97 | "mention" | "money" | "unknown"

```

Tag

Ontotext’s Tag API provides the ability to semantically enrich content of your choice with annotations by discovering mentions of both known and novel concepts.

Based on data from DBpedia and Wikidata, and processed with smart machine learning algorithms, it recognizes mentions of entities such as Person, Organisation, and Location, various relationships between them, as well as general topics and key phrases mentioned. Visit the NOW demonstrator to explore such entities found in news.
Create a TAG client

```
PREFIX ttxt: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
INSERT DATA {
  txtm-inst:tagService ttxt:connect txtm:Ces;
}
```

Find Tag entities through GraphDB

```
PREFIX ttxt: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
WHERE {
  ?searchDocument a txtm-inst:searchDocument;
  ttxt:service "http://www.ontotext.com/textmining-instance#";
  txtm:annotationText ?annotationText .
  ?annotation txtm-inst:annotationType ?annotationType ;
  txtm:annotationStart ?annotationStart ;
  txtm:annotationEnd ?annotationEnd ;
    ?item ?feature ?value
  } ^
}
```

For some annotations, an exact match to one or more IRIs in the knowledge graph are found and accessible through annotation features along with other annotation metadata.

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>IRI</th>
<th>Confidence</th>
<th>Mentioned Score</th>
<th>Retrieved Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td><a href="http://www.ontotext.com/textmining-instance#">http://www.ontotext.com/textmining-instance#</a></td>
<td>0.85</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>Person</td>
<td><a href="http://www.ontotext.com/textmining-instance#">http://www.ontotext.com/textmining-instance#</a></td>
<td>0.85</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>Organization</td>
<td><a href="http://www.ontotext.com/textmining-instance#">http://www.ontotext.com/textmining-instance#</a></td>
<td>0.85</td>
<td>0.95</td>
<td>0.95</td>
</tr>
</tbody>
</table>

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).


• `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, i.e, from NER taggers for which there is no suitable concept in the knowledge graph. Note that generated does not mean that the annotation is not trusted.

• `txtm:relevanceScore`: A float number that represents the level of relevancy of the annotation to the target document.

• `txtm:confidence`: A float number that represents the confidence score for the annotation to be produced.

### Extract Tag entities as web annotation 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:header "Accept: application/vnd.ontotext.ces+json+ld";
    txtm:header "Content-type: application/vnd.ontotext.ces+json+ld".
}
```

The common model applied for all services is no longer used because you get the Tag response in RDF as 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` (this is the default if nothing is specified)
- `Content-type: application/vnd.ontotext.ces+json+ld` - `Accept: 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 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 the JSON-LD, the following document features are required. Note that they should be passed using the `txtm:features` predicate on `?annotatedDocument` and in this order:

```
```
Here is a sample query:

```sparql
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 receiving an invitation 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 }
}
```

You can also use the `txtm:rawInput` predicate to provide your own raw JSON-LD document. The query above will look as follows, and will return the same results:

```sparql
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 "$0Id": "resource:some-new-guid-for-the-annotated-document-resource",
  "@graph": [
    { "@id": "resource:some-new-guid-for-the-annotated-document-resource",
      "@type": "AnnotatedDocument",
      "@context": "http://www.ontotext.com/textmining/instance#"
    }
  ];
 txtm:rawInput "$0Id": "resource:some-new-guid-for-the-annotated-document-resource",
  "@type": "AnnotatedDocument",
  "@context": "http://www.ontotext.com/textmining/instance#"
}
```
The supported returned response formats are JSON and JSON-LD.

**Extract 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` must be `application/json`. The headers of the POST request are passed using the predicate `http://www.ontotext.com/textmining#header`. The request body is passed 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://endpoint.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;
  
  "confi
```
and to the following query for consuming the annotations:

```sql
PREFIX : <http://www.ontotext.com/textmining#>
PREFIX inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
WHERE {
  ?searchDocument inst:myService;
    :text "body" .
  graph inst:myService {
    ?annotation :annotationText ?annotationText ;
      :annotationType ?annotationType ;
      :annotationStart ?annotationStart ;
      :annotationEnd ?annotationEnd ;
    }
  }
}
```

If we want to extract annotations using another named entity recognition provider, we can do so by creating a client for such services by providing a JSLT transformation. The transformation will convert the JSON returned by the target service to a JSON model understandable for the text mining plugin. The target JSON should look like this:

```json
{
  "content": "",
  "sentences": [],
  "features": {},
  "annotations": [
    {
      "text": "Google",
      "type": "Company",
      "startOffset": 78,
      "endOffset": 84,
      "confidence": 0.0,
      "features": {}
    }
  ]
}
```

where the only required part is:

```json
{
  "annotations": [
    {
      "text": "Google",
      "type": "Company",
      "startOffset": 78,
      "endOffset": 84,
    }
  ]
}
```
Google Cloud Natural Language API

Google Cloud Natural Language’s 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. It also requires a token to access the API.

Create a Google Cloud Natural Language API client

```prefix
taxm: <http://www.ontotext.com/textmining#>
taxm-inst: <http://www.ontotext.com/textmining/instance#>
insert data {
  `txtm-inst:myGoogleService txtm:connect txtm:Provider;`
  `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
      }
    }])
  
  ```
  
  `}}
  ```
}
```

Extract entities from Google 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
taxm: <http://www.ontotext.com/textmining#>
taxm-inst: <http://www.ontotext.com/textmining/instance#>
value ?featureItem ?
value WHERE {
  searchDocument a txtm-inst:myGoogleService;
  txtm:text ""{
  "document":{
  "type":"PLAIN_TEXT",
  "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"
```
Refinitiv API

Refinitiv’s PermIDs are open, permanent, and universal identifiers where underlying attributes capture the context of the identity they each represent.
Create a Refinitiv API client

```sql
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 (.).key : (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
    []
  }
}.
```
Extract Refinitiv PermID entities

```sparql
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>

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.'.
  txtm:text '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 {
    ?annotation txtm:annotationType ?annotationType .
    OPTIONAL {
    }
  }
}
```

<table>
<thead>
<tr>
<th>searchDocument</th>
<th>annotation</th>
<th>annotationText</th>
<th>annotationType</th>
<th>annotationStart</th>
<th>annotationEnd</th>
<th>feature</th>
<th>value</th>
<th>featureItem</th>
<th>featureValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>&quot;Dyson Ltd.&quot;</td>
<td>&quot;Person&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>&quot;James&quot;</td>
<td>&quot;Person&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>&quot;moving&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>&quot;back&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>&quot;residency&quot;</td>
<td>&quot;noun&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>&quot;US$29 billion&quot;</td>
<td>&quot;number&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>&quot;announced&quot;</td>
<td>&quot;verb&quot;</td>
<td>&quot;2018&quot;</td>
<td>&quot;2018&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

In the following example:

```sparql
PREFIX : <http://www.ontotext.com/textmining#>
PREFIX inst: <http://www.ontotext.com/textmining/instance#>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
WHERE {
    ?searchDocument a inst:razor;
        :text ?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 {
        ?annotation :annotationText ?annotationText ;
            :annotationType ?annotationType ;
            :annotationStart ?annotationStart ;
            :annotationEnd ?annotationEnd ;
          ?item ?feature ?value
        }
    }
}"
```

Quotation marks are escaped as follows:

The Vikings said in a statement they were aware of the allegation and "will await the results of that hearing before having further comment."

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: ‘‘. This will form the values of the valid JSON that the plugin will send to the service.
Compare 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 to the `?annotationKey` variable an IRI for the annotation based on the text and offsets, meaning that regardless of the service that generated the annotation, the same pieces of text will have the same `?annotationKey` IRIs. This can be used 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, meaning that they refer to the same piece of text.

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
WHERE {
   BIND ("'"?query?'" '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 {
   ?spacyAnnotation txtm:annotationText ?annotationText ;
   txtm:annotationKey ?annotationKey;
   txtm:annotationType ?spacyType .
}
?searchDocument2 a txtm-inst:tagService;
   txtm:text ?text .
graph txtm-inst:tagService {
   ?tagAnnotation txtm:annotationText ?annotationText ;
   txtm:annotationKey ?annotationKey;
   txtm:annotationType ?tagType .
}
```

Which will return:

<table>
<thead>
<tr>
<th>spacyDocument</th>
<th>tagDocument</th>
<th>spacyAnnotation</th>
<th>tagAnnotation</th>
<th>spacyType</th>
<th>tagType</th>
<th>annotationKey</th>
<th>annotationText</th>
</tr>
</thead>
</table>

The IRIs generated by the text mining plugin have the following meaning:

  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.

– `<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:

```xml
<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111/localSpacy/4>
```


For example:

```xml
<http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847/annotation/102/111>
```

**Enrich 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.

```sparql
PREFIX ttxm: <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 ("'" as ?text)

  ?searchDocument a txtm-INST:tagService;
  ttxm:text ?text .
  graph txtm-INST:tagService {
    ?tagAnnotation txtm:features ?item .
    ?item ttxm:exactMatch ?value
  }
}
```

Which will return:

<table>
<thead>
<tr>
<th>subject</th>
<th>predicate</th>
<th>object</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847">http://www.ontotext.com/textmining/document/ffa3feed18dacea1c195492cc1c06847</a></td>
<td>ttxm:annotations</td>
<td></td>
<td>GraphDB Documentation, Release 10.0.0</td>
</tr>
<tr>
<td><a href="http://my.knowledge.graph.com/textmining">http://my.knowledge.graph.com/textmining</a></td>
<td>my-kg:mentions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Of course, the power of RDF allows you to construct any graph you want based on the response from the named entity recognition service.

### 5.13.14.3 Error handling

Let’s say you have multiple documents with content that you want to send for annotation, for example documents from your own knowledge graph. For the example to work, insert the following documents in your repository:

```sparql
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.

```sparql
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 {

  ?searchDocument a txtm-inst:localSpacy;
  txtm:text ?content;
  txtm:serviceErrors -1.
  graph txtm-inst:localSpacy {
    OPTIONAL {
    }
    OPTIONAL {
    }
  }
}
```

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 (`Ctrl+C` in the terminal...
where the Docker is running). The error message is returned as a document feature.

<table>
<thead>
<tr>
<th>context</th>
<th>a</th>
<th>annotation</th>
<th>text</th>
<th>errorFeature</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG29</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BG30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.13.14.4 Manage 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.

List all clients

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT * where {
  ?instance a txtm:Service .
}
```
Get configuration for a client

```
PREFIX txtm: <http://www.ontotext.com/textmining#>
PREFIX txtm-inst: <http://www.ontotext.com/textmining/instance#>
SELECT * WHERE {
}
```

Drop 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 "".
}
```

5.13.14.5 Monitor 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
```

5.13.15 Sequences plugin

5.13.15.1 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.

To deploy it, please follow the GitHub instructions.
5.13.15.2 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 for your sequence, for example http://example.com/my/seq1. 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:

```
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. For a workaround in the cluster, see here.

Note: Using the below examples will not work inside the GraphDB Workbench as they need to be executed in one single transaction, and if run one by one, they would be performed in separate transactions. See here how to execute them in one transaction.

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):
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 ?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 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. New data using the sequence values can be generated by the same INSERT and added to GraphDB.

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 - as new sequence values do not need to be exposed to the client.
- the GraphDB Workbench - as 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)
};

# 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
}
```

**Workaround for using sequence values on the client with the cluster**

If you need to process your sequence values on the client in a GraphDB 9.x cluster environment, you can create a single-node (i.e., not part of a cluster) worker repository to provide the sequences. It is most convenient to have that repository on the same GraphDB instance as your primary master repository.

Let’s call the master repository where you will store your data `master1` and the second worker repository where you will create and use your sequences `seqrepo1`.  

5.13. Plugins 385
Managing sequences

Execute all *create*, *drop*, and *reset* statements in `seqrepo1`.

The examples below assume that you have created a sequence `http://example.com/my/seq1`.

Using sequences on the client

1. First, you need to obtain one or more new sequence values from the repository `seqrepo1`:
   a. Start a transaction in `seqrepo1`.
   b. Prepare the sequences for use by executing this in the same transaction:

   ```sql
   PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
   INSERT DATA {
     [] seq:prepare []
   }
   ```
   c. Obtain one or more new sequence values from the sequence `http://example.com/my/seq1`:

   ```sql
   PREFIX seq: <http://www.ontotext.com/plugins/sequences#>
   PREFIX my: <http://example.com/my/>
   SELECT ?next {
     my:seq1 seq:nextValue ?next
   }
   ```
   d. Commit the transaction in `seqrepo1`.

2. Then you can process the obtained values on the client, generate new data, and insert it into the master repository `master1`:
   a. Start a transaction in `master1`.
   b. Insert data using the obtained sequence values.
   c. Commit the transaction in `master1`.

Handling backups

To always ensure data consistency with backups, follow this order:

- Backup
  1. Backup the master repository `master1` first.
  2. Backup the sequence repository `seqrepo1` second.
- Restore
  1. Restore the sequence repository `seqrepo1` first.
  2. Restore the master repository `master1` second.

An alternative would be to not back up the `seqrepo1` repository but simply recreate the repository and the sequence (or reset the sequence) with the next potential sequence value from the `master1` repository. Here is a sample query that retrieves the next potential value (which is equal to the last used value + 1):

```sql
PREFIX ent: <http://www.ontotext.com/owlim/entity#>
PREFIX my: <http://example.com/my/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?nextValue WHERE {
   ... (continues on next page)
Note that this example assumes that sequence values were used to generate IRIs, and IRIs with higher values were used for the first time after IRIs with lower values were used.

Activate & enable plugins Activating and deactivating plugins from the Workbench, and information about enabling/disabling some of them.

Plugin API A framework and a set of public classes and interfaces that allow developers to extend GraphDB in many useful ways.

RDF Rank An algorithm that identifies the more important or more popular entities in the repository by examining their interconnectedness.

Semantic similarity searches Exploring and searching semantic similarity in RDF resources.

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.

Provenance Tracking changes within the context of a transaction identified by a unique ID.

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 are user-defined indexes. They are created with SPARQL, and differ from the system indexes in that they can be configured dynamically at runtime. Each user with write access to a certain repository can define such an index. These plugins are:

Autocomplete index Suggestions for the IRIs’ local names in the SPARQL editor and the View Resource page.

GeoSPARQL support GeoSPARQL is 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.

Sequences plugin Providing transactional sequences for GraphDB. A sequence is a long counter that can be atomically increment ed in a transaction to provide incremental IDs.

The GraphDB Connectors are such indexes as well.
5.14 GraphDB Connectors

5.14.1 Elasticsearch GraphDB connector

Note: This feature requires a GraphDB Enterprise license.

5.14.1.1 Overview and features

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Elasticsearch but have 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 defined as a sequence of triples where each triple’s object is the subject of the following triple.

The main features of the GraphDB Connectors are:

- maintaining an index that is always in sync with the data stored in GraphDB;
- multiple independent instances per repository;
- the entities for synchronization are 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's 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.

5.14.1.2 Usage

All interactions with the Elasticsearch GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

- INSERT for creating and deleting connector instances;
- SELECT for listing connector instances and querying their configuration parameters;
- INSERT/SELECT for storing and querying data as part of the normal GraphDB data workflow.

In general, this corresponds to INSERT adds or modifies data and SELECT queries existing data.
Each connector implementation defines its own IRI prefix 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 this prefix, e.g., 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 prefix to avoid clashing with any of the command predicates. For Elasticsearch, the instance prefix is http://www.ontotext.com/connectors/elasticsearch/instance#.

Sample data All examples use the following sample data that describes five fictitious wines: Yowowine, 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.

```r
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2000/01/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wine: <http://www.ontotext.com/example/wine#> .

wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlo rdfs:type wine:Grape ; rdfs:label "Merlo" .

wine:Merlot rdfs:type wine:Grape ; rdfs:label "Merlot" .


wine:CabernetFranc rdfs:type wine:Grape ; rdfs:label "Cabernet Franc" .


wine:Chardonnay rdfs:type wine:Grape ; rdfs:label "Chardonnay" .

wine:Yowowine rdfs:type wine:RedWine ; wine:madeFromGrape wine:Merlo ;
                           wine:hasSugar "dry" ;
                           wine:hasYear "2013"^^xsd:integer .

wine:Franvino rdfs:type wine:RedWine ; wine:madeFromGrape wine:Merlot ;
                   wine:madeFromGrape wine:CabernetFranc ;
                   wine:hasSugar "dry" ;
                   wine:hasYear "2012"^^xsd:integer .

wine:Noirette rdfs:type wine:RedWine ;
               wine:madeFromGrape wine:PinotNoir ;
               wine:hasSugar "medium" ;
               wine:hasYear "2012"^^xsd:integer .

wine:Blanquito rdfs:type wine:WhiteWine ;
```

(continues on next page)
5.14.1.3 Setup and maintenance

Prerequisites

Third-party component versions  This version of the Elasticsearch GraphDB Connector uses Elasticsearch version 7.16.3.

Tip: Since version 2.0, by default Elasticsearch commits the translog at the end of every index, delete, update, or bulk request. The new configuration may cause a massive slowdown of the Elasticsearch connector, so we highly recommend to change the index.translog.durability value to async. For more information, see Elasticsearch's transaction log settings.

Tip: In Elasticsearch 7.x.x, the default value for the wait_for_active_shards parameter of the open index command has been changed from 0 to 1. This means that the command will now by default wait for all primary shards of the opened index to be allocated. You can find more information about it here. Depending on your specific case, you can experiment with different values to find the optimal ones for you, for example: "indexCreateSettings": {"number_of_shards" : 5, "number_of_replicas" : 1, "write.wait_for_active_shards" : 0}.

Creating a connector instance

Creating a connector instance is done by sending a SPARQL query with the following configuration data:

- the name of the connector instance (e.g., my_index);
- an Elasticsearch instance to synchronize to;
- classes to synchronize;
- properties to synchronize.

The configuration data has to be provided as a JSON string representation and passed together with the create command.

If you create the connector via the Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.
Using the 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.
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.

Create new Elasticsearch Connector

Language: en

Create new Elasticsearch Connector

Language: en
Using the create command

The create command is triggered by a SPARQL INSERT with the createConnector predicate, e.g., it creates a connector instance called my_index, which synchronizes the wines from the sample data above:

```sparql
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

INSERT DATA {
  \n  elastic-index:my_index elastic:createConnector '"
  { 
    "elasticsearchNode": "localhost:9200",
    "types": [ 
      "http://www.ontotext.com/example/wine#Wine"
    ],
    "fields": [ 
      { 
        "fieldName": "grape",
        "propertyChain": [ 
          "http://www.ontotext.com/example/wine#madeFromGrape",
          "http://www.w3.org/2000/01/rdf-schema#label"
        ]
      },
      { 
        "fieldName": "sugar",
        "propertyChain": [ 
          "http://www.ontotext.com/example/wine#hasSugar"
        ],
        "analyzed": false
      },
      { 
        "fieldName": "year",
        "propertyChain": [ 
          "http://www.ontotext.com/example/wine#hasYear"
        ],
        "analyzed": false
      }
    ]
  }
}
```
GraphDB Documentation, Release 10.0.0

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 and, in the example, it is only entities of the type `http://www.ontotext.com/example/wine#Wine` (and its subtypes). The "fields" key defines the mapping from RDF to Elasticsearch. The basic building block is the property chain, i.e., 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 later used in the queries.

Grape is an example of a property chain composed of more than one property. First, we take the wine’s made-FromGrape property, the object of which is an instance of the type Grape, and then we take the rdfs:label of this instance. 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, 2013, and thus 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.ontotext.com/example/wine#Wine"],
  "fields": [
    {
      "fieldName": "grape",
      "propertyChain": ["http://www.ontotext.com/example/wine#madeFromGrape",
                        "http://www.w3.org/2000/01/rdf-schema#label"
      ]
    }
  ]
}
```

(continues on next page)
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:

<table>
<thead>
<tr>
<th>field name</th>
<th>Elasticsearch config</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_graphdb_id</code></td>
<td><code>&quot;type&quot;:&quot;long&quot;, &quot;index&quot;:true, &quot;store&quot;:true</code></td>
</tr>
<tr>
<td>grape</td>
<td><code>&quot;type&quot;:&quot;text&quot;, &quot;index&quot;:true, &quot;store&quot;:true</code></td>
</tr>
<tr>
<td>sugar</td>
<td><code>&quot;type&quot;:&quot;keyword&quot;, &quot;index&quot;:true, &quot;store&quot;:true</code></td>
</tr>
<tr>
<td>year</td>
<td><code>&quot;type&quot;:&quot;keyword&quot;, &quot;index&quot;:true, &quot;store&quot;:true</code></td>
</tr>
</tbody>
</table>

_{-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.

**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 `dropConnector` predicate where the name of the connector instance has to be in the subject position, e.g., this removes the connector `my_index`:

```sparql
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
INSERT DATA {

(continues on next page)
You can also force drop a connector in case a normal delete 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, and check Force delete in the dialog box.

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:

```sparql
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 .
}
```

Listing available connector instances

In the Connectors management view

Existing Connector instances show under Existing connectors (below the New Connector button). Click the name of an instance to view its configuration and SPARQL query, or click the repair / delete icons to perform these operations.
With a SPARQL query

Listing connector instances returns all previously created instances. It is a SELECT query with the `listConnectors` predicate:

```sparql
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?cntUri ?cntStr {  
  ?cntUri elastic:listConnectors ?cntStr .
}
```

`?cntUri` is bound to the prefixed IRI of the connector instance that was used during creation, e.g., `http://www.ontotext.com/connectors/elasticsearch/instance#my_index`, while `?cntStr` is bound to a string, representing the part after the prefix, e.g., "my_index".
Instance status check

The internal state of each connector instance can be queried using a SELECT query and the `connectorStatus` predicate:

```sparql
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>

SELECT ?cntUri ?cntStatus {
}
```

`?cntUri` is bound to the prefixed 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.

5.14.1.4 Working with data

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, i.e., you must simply execute standard SPARQL `INSERT/DELETE` queries. This is achieved by intercepting all changes in the plugin and determining which abstract documents need to be updated.

Simple queries

Once a connector instance has been created, it is possible to query data from it through SPARQL. For each matching abstract document, the connector instance returns the document subject. In its simplest form, querying is achieved by using a SELECT and providing the Elasticsearch query as the object of the `query` predicate:

```sparql
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 binds `?entity` to the two wines made from grapes that have “cabernet” in their name, namely :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.

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 :query.
3. Request the matching entities through the :entities predicate.

It is also possible to provide per-query search options by using one or more option predicates. The option predicates are described in detail below.

5.14. GraphDB Connectors
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 here, execute the following query:

```sparql
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" : {
    "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

The bound ?entity can be used in other SPARQL triples in order to build complex queries that fetch additional data from GraphDB, for example, to see the actual grapes in the matching wines as well as the year they were made:

```sparql
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>
PREFIX wine: <http://www.ontotext.com/example/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:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?grape</th>
<th>?year</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>:CabernetSauvignon</td>
<td>2013</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:Merlot</td>
<td>2012</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:CabernetFranc</td>
<td>2012</td>
</tr>
</tbody>
</table>
**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:

```sparql
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:

<table>
<thead>
<tr>
<th>entity</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>0.9442660212516785</td>
</tr>
<tr>
<td>:Franvino</td>
<td>0.7554128170013428</td>
</tr>
</tbody>
</table>

### Basic facet queries

Consider the sample wine data and the my_index connector instance described previously. You can also query facets using the same instance:

```sparql
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 _:f .
  _:f elastic:facetName ?facetName .
  _:f elastic:facetValue ?facetValue .
  _:f 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 facets predicate. As each facet has three components (name, value, and count), the facets predicate binds a blank node, which in turn 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:

<table>
<thead>
<tr>
<th>facetName</th>
<th>facetValue</th>
<th>facetsCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>2012</td>
<td>3</td>
</tr>
<tr>
<td>year</td>
<td>2013</td>
<td>2</td>
</tr>
<tr>
<td>sugar</td>
<td>dry</td>
<td>3</td>
</tr>
<tr>
<td>sugar</td>
<td>medium</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

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:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
<th>Elasticsearch counterpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>:name</td>
<td>Bucket name</td>
<td>getName()</td>
</tr>
<tr>
<td>:key</td>
<td>Key or value associated with the bucket</td>
<td>getValue() or getKey()</td>
</tr>
<tr>
<td>:count</td>
<td>Count of documents in the bucket</td>
<td>getDocCount(), getValue()</td>
</tr>
<tr>
<td>:from</td>
<td>Start of range</td>
<td>getFrom(), getFromAsDate()</td>
</tr>
<tr>
<td>:to</td>
<td>End of range (RangeFacet)</td>
<td>getTo(), getToAsDate()</td>
</tr>
<tr>
<td>:min</td>
<td>Minimum value</td>
<td>getMin(), getValue()</td>
</tr>
<tr>
<td>:max</td>
<td>Maximum value</td>
<td>getMax(), getValue()</td>
</tr>
<tr>
<td>:sum</td>
<td>Sum value</td>
<td>getSum(), getValue()</td>
</tr>
<tr>
<td>:avg</td>
<td>Average value</td>
<td>getAvg(), getValue()</td>
</tr>
<tr>
<td>:sum_of_squares</td>
<td>Sum of squares value</td>
<td>getSumOfSquares()</td>
</tr>
<tr>
<td>:variance</td>
<td>Variance value</td>
<td>getVariance()</td>
</tr>
<tr>
<td>:std_deviation</td>
<td>Standard deviation value</td>
<td>getStdDeviation()</td>
</tr>
<tr>
<td>:parent</td>
<td>Sub-aggregations: points to the parent (upper level) blank node</td>
<td></td>
</tr>
<tr>
<td>:level</td>
<td>Sub-aggregations: level number where 1 is the uppermost level and the following levels are 2, 3 and so on</td>
<td>getKey() or getValue()</td>
</tr>
<tr>
<td>:levelName</td>
<td>Sub-aggregations: level name</td>
<td>getKey() or getValue()</td>
</tr>
</tbody>
</table>
Sorting

It is possible to sort the entities returned by a connector query according to one or more fields. Sorting is achieved by the `orderBy` predicate the value of which is a comma-delimited list of fields. Each field can be prefixed with a minus to indicate sorting in descending order. For example:

```sparql
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 "year:2013" ;
  elastic:orderBy "-sugar" ;
  elastic:entities ?entity .
}
```

The result contains wines produced in 2013 sorted according to their sugar content in descending order:

```
entity
Rozova
Yoyowine
```

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. To remedy this, use `ORDER BY` from SPARQL.

**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.

Limit and offset

Limit and offset are supported on the Elasticsearch side of the query. This is achieved through the predicates `limit` and `offset`. Consider this example in which an offset of 1 and a limit of 1 are specified:

```sparql
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 .
}
```

The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:
Note: The specific order in which GraphDB returns the results depends on how Elasticsearch returns the matches, unless sorting is specified.

Snippet extraction

Snippet extraction is used for extracting highlighted snippets of text that match the query. The snippets are accessed through the dedicated predicate `snippets`. It binds a blank node that in turn provides the actual snippets via the predicates `snippetField` and `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 .
  ?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:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?snippetField</th>
<th>?snippetText</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt; Sauvignon</td>
</tr>
<tr>
<td>:Franvino</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt; Franc</td>
</tr>
</tbody>
</table>

Note: The actual snippets might be different as this depends on the specific Elasticsearch implementation.

It is possible to tweak how the snippets are collected/composed by using the following option predicates:

- :snippetSize - sets the maximum size of the extracted text fragment, 250 by default;
- :snippetSpanOpen - the text to insert before the highlighted text, `<em>` by default;
- :snippetSpanClose - the text to insert after the highlighted text, `</em>` by default.

The option predicates are set on the query instance, much like the :query predicate.
Snippets from nested documents

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”:

```turtle
PREFIX elastic: <http://www.ontotext.com/connectors/elasticsearch#>
PREFIX elastic-index: <http://www.ontotext.com/connectors/elasticsearch/instance#>

SELECT ?entity ?snippetInnerField ?snippetField ?snippetText
WHERE {
  ?search a elastic-index:my_index ;
  elastic:query ""
  {
    "query": {
      "nested": {
        "path": "grandChildren",
        "query": {
          "bool": {
            "must": {
              "match": {
                "grandChildren.name": "tylor"
              }
            }
          }
        }
      }
    }
    "..."
  }
  elastic:entities ?entity .
  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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>urn:Eva</td>
<td>grandChildren</td>
<td>grandChildren.name</td>
<td>John-&lt;em&gt;Tylor&lt;/em&gt;</td>
</tr>
<tr>
<td>urn:John</td>
<td>grandChildren</td>
<td>grandChildren.name</td>
<td>John-&lt;em&gt;Tylor&lt;/em&gt;</td>
</tr>
<tr>
<td>urn:John</td>
<td>grandChildren</td>
<td>grandChildren.name</td>
<td>&lt;em&gt;Tylor&lt;/em&gt;</td>
</tr>
<tr>
<td>urn:Mary</td>
<td>grandChildren</td>
<td>grandChildren.name</td>
<td>&lt;em&gt;Tylor&lt;/em&gt;</td>
</tr>
</tbody>
</table>

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.
Total hits

You can get the total number of hits by using the `totalHits` predicate, e.g., for the connector instance `my_index` and a query that retrieves all wines made in 2012:

```sparql
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`.

5.14.1.5 List of creation parameters

The creation parameters define how a connector instance is created by the `:createConnector` predicate. 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 index for tracking updates to chains)

The only way to index changes in data after the connector has been created is to repair (or drop/recreate) the connector.

`importGraph` (boolean), optional, a virtual graph containing data from which to create the connector

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 `es:graph`, where the `PREFIX es: <http://www.ontotext.com/connectors/elasticsearch#>` control prefix is used. The data have to 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 GDB Workbench, this can be done by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This functionality requires `read-only mode`.

```sparql
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 {
  elastic-index:my_index elastic:createConnector ' {
"readonly": true,
"importGraph": true,
"fields": []
}
```
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 functionality requires readonly mode.

detectFields (boolean), optional, detects fields   This mode introduces automatic field detection when creating a connector. Instead of providing fields: [ ... ] in the JSON, you can skip this and get automatic fields where each field will have a single predicate chain and its field name will be the same as the predicate.

In this mode, specifying types is optional too. 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 the Workbench.

elasticsearchMode (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 format of the node value is of the form http://hostname.domain:port, https:// is allowed too. No default value. Can be updated at runtime without having 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 without having 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 without having to rebuild the index.

elasticsearchClusterSniff (boolean), controls whether to build the server address list by sniffing on the Elasticsearch cluster   Corresponds to the Elasticsearch client.transport.sniff option. True by default. Can be updated at runtime without having 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 without having to rebuild the index.

bulkUpdateRequestSize (integer), controls the maximum size in bytes per bulk request   Defaults to 5,242,880 bytes (5 million bytes). Can be updated at runtime without having to rebuild the index.

The limits of bulkUpdateBatchSize and bulkUpdateRequestSize are combined, and a bulk request is sent once either limit is hit.

authenticationConfiguratorClass optional, provides custom authentication behavior

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. This can be done 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. In addition, an empty range can be used 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 to keep the field names simple.

- **fieldNameTransform (one of none, predicate or predicate.localName), optional, none by default** The field name transform defines an optional transformation of the field name.
  - none: The field name is supplied via the `fieldName` option.
  - predicate: The field name is derived from the full IRI of the last predicate of the chain, e.g., if the last predicate was http://www.w3.org/2000/01/rdf-schema#label, then the field name will be http://www.w3.org/2000/01/rdf-schema#label too.
  - predicate.localName: The field name is the derived from the local name of the IRI of the last predicate of the chain, e.g., 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 defined as 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** The default value (defaultValue) provides means for specifying 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  Fields can be stored in Elasticsearch, and this is controlled by the Boolean option *stored*. Stored fields are required for retrieving snippets. 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 (they 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 that 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. Fielddata 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.

  The mapping can be overridden through the property "datatype", 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:, e.g., both xsd:long and native:long map to the long type in Elasticsearch.

• **nativeSettings** (json), optional, custom field settings  The setting for the Elasticsearch mapping parameters of the respective field, for example 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, provide 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.ontotext.com/example/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.

## Updating parameters at runtime

As mentioned above, the following connector parameters can be updated at runtime without having to rebuild the index:

- `elasticsearchNode`
- `elasticsearchClusterSniff`
- `elasticsearchBasicAuthUser`
- `elasticsearchBasicAuthPassword`
- `bulkUpdateBatchSize`
- `bulkUpdateRequestSize`

This can be done by executing the following SPARQL update, here with examples for changing the user and password:

```sparql
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"
  } .
} .
```
Special field definitions

Nested objects

Nested objects are Elasticsearch documents that are used as values in the main document or other nested objects (up to five levels of nesting is possible). They are defined with the `objectFields` option.

Having the following data consisting of children and grand children relations:

```xml
<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:Billy> .

<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)
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 as we want to query the nested objects independently of each other:

```sql
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.ontotext.com/example#gadget"],
    "fields": [
      {
        "fieldName": "name",
        "propertyChain": ["urn:name"
      },
      {
        "fieldName": "age",
        "propertyChain": ["urn:age"
      },
      {
        "fieldName": "hasSpouse",
        "propertyChain": ["urn:hasSpouse"
      },
      {
        "fieldName": "gender",
        "propertyChain": ["urn:gender",
          "urn:label"
        }
      },
      {
        "fieldName": "children",
        "propertyChain": ["urn:hasChild"
      },
      {
        "fieldName": "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",
"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"}
To find male grandchildren age of 5 years and older, we will use the following query:

```sql
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" : {
      "nested" : {
        "path" : "grandChildren",
        "query" : {
          "bool" : [
            "must" : [
              {
                "match" : {
                  "grandChildren.gender" : "male"
                },
              },
              {
                "range" : {
                  "grandChildren.age" : {
                    "gt" : 5
                  }
                }
              }
            ]
          }
        }
      }
    }
  }
}
```

(continues on next page)
The result looks like this:

```
?entity
urn:Eva
urn:John
```

Copy fields

Often, it is convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases, e.g., faceting or sorting vs full-text search. The Elasticsearch GraphDB Connector has explicit support for fields that copy their value from another field. This is achieved by specifying 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",
    "propertyChain": [
      "http://www.ontotext.com/example/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, e.g., 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 alphanumeric sequence of convenience. 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"
 }
 ...]
... 
```

The values of the fields name$1$ and name$2$ will be merged and synchronized to the field name in Elasticsearch.

**Note:** You cannot mix suffixed and unsuffixed fields with the same same, e.g., if you defined myField$new$ and myField$old$, you cannot have a field called just myField.

Filters and fields with multiple property chains

Filters can be used 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, e.g., ?myField
- Virtual fields are specified with the suffix, e.g., ?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, where the 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 ""
}
```

(continues on next page)
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**

The named graph of a given value can be indexed 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.

```json
{  
  "elasticsearchNode": "localhost:9200",
  "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 named graph of each value of the property http://www.ontotext.com/example#name into the field nameGraph.
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:

```json
{  
  "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 (e.g., [http://www.ontotext.com/example/wine#Franvino](http://www.ontotext.com/example/wine#Franvino) from our small example dataset) indexed as a regular field. This can be achieved by specifying a property chain with a single property referring to the pseudo-IRI $self. For example:

```prefix
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.ontotext.com/example/wine#Wine"  
    ],  
    "fields": [  
      {  
        "fieldName": "entityId",  
        "propertyChain": [  
          "$self"  
        ],  
      },  
      {  
        "fieldName": "grape",  
        "propertyChain": [  
          "http://www.ontotext.com/example/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*. 

---

**Chapter 5. Usage**
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`.

5.14.1.6 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:

<table>
<thead>
<tr>
<th>RDF value</th>
<th>RDF datatype</th>
<th>Elasticsearch type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRI</td>
<td>n/a</td>
<td>keyword</td>
</tr>
<tr>
<td>literal</td>
<td>any type not explicitly mentioned below</td>
<td>text</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:double</td>
<td>double</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:float</td>
<td>float</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:long</td>
<td>long</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:int</td>
<td>integer</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:dateTime</td>
<td>date with format: strict_date_time</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:date</td>
<td>date with format: strict_date</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:time</td>
<td>date with format: strict_time_no_millis</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYear</td>
<td>date with format: strict_year</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYearMonth</td>
<td>date with format: strict_year_month</td>
</tr>
</tbody>
</table>

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, e.g., the first value has no datatype but other values have.

It is therefore recommended to set `datatype` to a fixed value, e.g. `xsd:date`.

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 format and are `proleptic` years, i.e., 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, we define the undetermined time zone as UTC, i.e., "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. E.g., "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 strict timezone adherence is followed. We have chosen to ignore the timezone on any values that do not have an associated time value, e.g.:

- "2020-02-15+02:00"^^xsd:date
- "2020-02+02:00"^^xsd:gYearMonth
- "2020+02:00"^^xsd:gYear

All of the above will be treated as if they specified UTC as their timezone.

5.14.1.7 Entity filtering

The Elasticsearch connector supports four kinds of entity filters used to fine-tune the set of entities and/or 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 FILTER() inside a SPARQL query but not exactly the same. In them, each configured field can be referred to by prefixing it with a ?, much like referring to a variable in SPARQL.

**Types of filters**

**Top-level value filter** The top-level value filter is specified via `valueFilter`. 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 and 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 via `documentFilter`. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. It can be used to enforce global document restrictions, e.g., 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 via `valueFilter` 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 `$this`. Other field names can be used to filter the current field’s value based on the value of another field, e.g., `$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 via `documentFilter 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 we have a field `children` that defines a nested document, and we use a filter like `?age < "10"^^xsd:int`, we will be referring to the field `children.age`. We 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, `Souter.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.*

**Filter operators**

The filter operators are used to test if the value of a given field satisfies some condition:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?var in (value1, value2, ...)</code></td>
<td>Tests if the field <code>var</code>’s value is one of the specified values. Values are compared strictly unlike the similar SPARQL operator, i.e. for literals to match their datatype must be exactly the same (similar to how SPARQL <code>sameTerm</code> works). Values that do not match, are treated as if they were not present in the repository.  &lt;br&gt;Example:  &lt;br&gt;<code>?status in (&quot;active&quot;, &quot;new&quot;)</code></td>
</tr>
<tr>
<td><code>?var not in (value1, value2, ...)</code></td>
<td>The negated version of the <code>in</code> operator.  &lt;br&gt;Example:  &lt;br&gt;<code>?status not in (&quot;archived&quot;)</code></td>
</tr>
<tr>
<td><code>bound(?var)</code></td>
<td>Tests if the field <code>var</code> has a valid value. This can be used to make the field compulsory.  &lt;br&gt;Example:  &lt;br&gt;<code>bound(?name)</code></td>
</tr>
<tr>
<td><code>isExplicit(?var)</code></td>
<td>Tests if the field <code>var</code>’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 <code>parent(?var)</code> as many times as needed.  &lt;br&gt;Example:  &lt;br&gt;<code>isExplicit(?name)</code></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?var = value (equal to)</td>
<td>RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The field var’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’s value is &quot;150&quot;^^xsd:int and dateOfBirth’s value is &quot;1990-01-01&quot;^^xsd:date, then:</td>
</tr>
</tbody>
</table>
| ?var != value (not equal to) | ?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 |
| ?var > value (greater than) | regex(?var, "pattern")
regex(?var, "pattern", "i") | Tests if the field var’s value matches the given regular expression pattern. If the "i" flag option is present, this indicates that the match operates in case-insensitive mode. Values that do not match are treated as if they were not present in the repository. Example: regex(?name, "^mrs?", "i") |
| ?var >= value (greater than or equal to) | expr1 || expr2
expr1 or expr2 | Logical disjunction of expressions expr1 and expr2. Examples: bound(?name) || bound(?company)
bound(?name) or bound(?company) |
| ?var < value (less than) | expr1 & expr2
expr1 and expr2 | Logical conjunction of expressions expr1 and expr2. Examples: bound(?status) && ?status in ("active", "new")
bound(?status) and ?status in ("active", "new") |
| ?var <= value (less than or equal to) | !expr | Logical negation of expression expr. Example: !bound(?company) |

Continued on next page
Filter modifiers

In addition to the operators, there are some constructions that can be used to 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)` is used for going to a previous level in a property chain. It can be applied recursively as many times as needed, e.g., `parent(parent(parent(?var)))` goes back in the chain three times. The effective value of `parent(?var)` can be used with the `in` or `not in` operator like this: `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator like this: `parent(bound(?var))`.

Accessing an element beyond the chain The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) is used to access additional values that are accessible through the property `uri`. In essence, this construction corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the field `var`. The effective value of `?var -> uri` can be used with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. It can be combined with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. The same construction can be applied to the `bound` operator like this: `bound(??company) -> <urn:hasBranch>`, or even combined with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (alternatively, 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)` is used for accessing 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(?x)` is the recommended way.

The construction can be combined with `parent()` like this: `graph(parent(?a)) in (<urn:a>).`

Filtering by language tags The construction `lang(?var)` is used for accessing 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")`. The construction can be combined with `parent()` and an element beyond the chain like this: `lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg")`. Literal values without language tags can be filtered by using an empty tag: `""`.

Current context variable $this The special field variable $this (and not `?this`, `$?this`, `$?this`) is used to refer to the current context. In the top-level value filter and the top-level document filter it refers to the document ID. 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 ID.

`ALL()` quantifier In the context of document-level filtering, a match is true if at least one of potentially many field values match, e.g., `?location = <urn:Europe>` would return true if the document contains `{ "location": ["<urn:Asia>", "<urn:Europe>"] }`.

In addition to this, you can also use the `ALL()` quantifier when you need all values to match, e.g., `ALL(?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

Entity filters and default values Entity filters can be combined with default values in order to get more flexible behavior.

A typical use-case for an entity filter is having soft deletes, i.e., instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.
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 comparing the field value to 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 as 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 being indexed first.

Basic entity filter example

Given the following RDF data:

```rdfs
@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 synchronised 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 Synless" ;
  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.ontotext.com/example#gadget"],
  "fields": [
  
  (continues on next page)
The entity :beta is not synchronized as 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.

**Advanced entity filter example**

Sometimes, data represented in RDF is not well suited to map directly to non-RDF. For example, if you have news articles and they can be tagged with different concepts (locations, persons, events, etc.), 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 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)
Now, if you map this data to Elasticsearch so that the property :taggedWith is mapped to separate fields taggedWithPerson and taggedWithLocation according to the type of x (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.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": "$this -> type in (<http://www.ontotext.com/example2#Person>)"
  },
  {
"fieldName": "taggedWithLocation",
"propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
"valueFilter": "$this -> type in (<http://www.ontotext.com/example2#Location>)"
  }
]
...
}
```
The six articles in the RDF data above will be mapped as such:

<table>
<thead>
<tr>
<th>Article IRI</th>
<th>Value in taggedWith-Person</th>
<th>Value in taggedWithLocation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Article1</td>
<td>:Einstein</td>
<td>:Berlin</td>
<td>: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 as it does not match the filter.</td>
</tr>
<tr>
<td>:Article2</td>
<td></td>
<td>:Berlin</td>
<td>:taggedWith has the value :Berlin. After the filter is applied, only taggedWithLocation is populated.</td>
</tr>
<tr>
<td>:Article3</td>
<td>:Mozart</td>
<td></td>
<td>:taggedWith has the value :Mozart. After the filter is applied, only taggedWithPerson is populated.</td>
</tr>
<tr>
<td>:Article4</td>
<td>:Mozart</td>
<td>:Berlin</td>
<td>:taggedWith has the values :Berlin and :Mozart. The filter leaves only the correct values in the respective fields.</td>
</tr>
<tr>
<td>:Article5</td>
<td></td>
<td></td>
<td>:taggedWith has no values. The filter is not relevant.</td>
</tr>
<tr>
<td>:Article6</td>
<td></td>
<td></td>
<td>:taggedWith has the value :Cannes-FF. The filter removes it as it does not match.</td>
</tr>
</tbody>
</table>

This can be checked 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 _:f .
  _:f elastic:facetName ?facetName ;
  elastic:facetValue ?facetValue ;
}
```

If the filter was applied, you should get only :Berlin for taggedWithLocation and only :Einstein and :Mozart for taggedWithPerson:

<table>
<thead>
<tr>
<th>?facetName</th>
<th>?facetValue</th>
<th>?facetCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>taggedWithLocation</td>
<td><a href="http://www.ontotext.com/example2#Berlin">http://www.ontotext.com/example2#Berlin</a></td>
<td>3</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Mozart">http://www.ontotext.com/example2#Mozart</a></td>
<td>2</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Einstein">http://www.ontotext.com/example2#Einstein</a></td>
<td>1</td>
</tr>
</tbody>
</table>
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)

5.14.1.8 Overview of connector predicates

The following diagram shows a summary of all predicates that can administrate (create, drop, check status) 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.
5.14.1.9 Caveats

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 need to come before any predicates that fetch results.

The diagram in Overview of connector predicates provides a quick overview of the predicates.

5.14.1.10 Upgrading from previous versions

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.

5.14.2 Lucene GraphDB connector

5.14.2.1 Overview and features

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Lucene but have 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 (a 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 defined as a sequence of triples where each triple’s object is the subject of the following triple.
The main features of the GraphDB Connectors are:

- maintaining an index that is always in sync with the data stored in GraphDB;
- multiple independent instances per repository;
- the entities for synchronization are 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's 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 Lucen types;
- specifying which Lucene analyzer to use (the default is Lucene’s StandardAnalyzer);
- stripping HTML/XML tags in literals (the default is not to strip markup);
- boosting an entity by the numeric value of one or more predicates;
- custom scoring expressions at query time to evaluate score based on Lucene score and entity boost.

Each feature is described in detail below.

5.14.2.2 Usage

All interactions with the Lucene GraphDB Connector shall be done through SPARQL queries.

There are three types of SPARQL queries:

- INSERT for creating and deleting connector instances;
- SELECT for listing connector instances and querying their configuration parameters;
- INSERT/SELECT for storing and querying data as part of the normal GraphDB data workflow.

In general, this corresponds to INSERT adds or modifies data and SELECT queries existing data.

Each connector implementation defines its own IRI prefix 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 this prefix, e.g., 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 prefix to avoid clashing with any of the command predicates. For Lucene, the instance prefix is http://www.ontotext.com/connectors/lucene/instance#.

Sample data All examples use the following sample data that describes five fictitious wines: Yoyowine, Fran-vino, Noirette, Blanquito and Rozova as well as the grape varieties required to make these wines. The minimum required ruleset level in GraphDB is RDFS.

```text
@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.ontotext.com/example/wine#> .
```

(continues on next page)
<table>
<thead>
<tr>
<th>Wine Type</th>
<th>SubClassOf</th>
<th>Grape Type</th>
<th>Sugar</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Wine</td>
<td>wine:Wine</td>
<td>wine:Grape</td>
<td>dry</td>
<td>2013</td>
</tr>
<tr>
<td>White Wine</td>
<td>wine:Wine</td>
<td>wine:Grape</td>
<td>dry</td>
<td>2012</td>
</tr>
<tr>
<td>Rose Wine</td>
<td>wine:Wine</td>
<td>wine:Grape</td>
<td>medium</td>
<td>2012</td>
</tr>
</tbody>
</table>

Merlot:
- `rdf:type` wine:Grape
- `rdfs:label` "Merlot"

Cabernet Sauvignon:
- `rdf:type` wine:Grape
- `rdfs:label"Cabernet Sauvignon"

Cabernet Franc:
- `rdf:type` wine:Grape
- `rdfs:label"Cabernet Franc"

Pinot Noir:
- `rdf:type` wine:Grape
- `rdfs:label"Pinot Noir"

Chardonnay:
- `rdf:type` wine:Grape
- `rdfs:label"Chardonnay"

Yoyo Wine:
- `rdf:type` wine:RedWine
- `wine:madeFromGrape` wine:Cabernet Sauvignon
- `wine:hasSugar"dry"
- `wine:hasYear"2013"^^xsd:integer

Franvino:
- `rdf:type` wine:RedWine
- `wine:madeFromGrape` wine:Merlo
- `wine:madeFromGrape` wine:Cabernet Franc
- `wine:hasSugar"dry"
- `wine:hasYear"2012"^^xsd:integer

Noirette:
- `rdf:type` wine:RedWine
- `wine:madeFromGrape` wine:Pinot Noir
- `wine:hasSugar"medium"
- `wine:hasYear"2012"^^xsd:integer

Blanquito:
- `rdf:type` wine:WhiteWine
- `wine:madeFromGrape` wine:Chardonnay
- `wine:hasSugar"dry"
- `wine:hasYear"2012"^^xsd:integer

Rozova:
- `rdf:type` wine:RoseWine
- `wine:madeFromGrape` wine:Pinot Noir
- `wine:hasSugar"medium"
- `wine:hasYear"2013"^^xsd:integer

5.14.2.3 Setup and maintenance

**Third-party component versions**  This version of the Lucene GraphDB Connector uses Lucene version 8.11.1.

Creating a connector instance

Creating a connector instance is done by sending a SPARQL query with the following configuration data:

- the name of the connector instance (e.g., `my_index`);
- classes to synchronize;
- properties to synchronize.

The configuration data has to be provided as a JSON string representation and passed together with the create command.

If you create the connector via the Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

**Using the 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.
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.
### Create new Lucene Connector

<table>
<thead>
<tr>
<th>Name</th>
<th>my_index</th>
</tr>
</thead>
</table>

#### Fields

<table>
<thead>
<tr>
<th>Field name</th>
<th>grape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property chain</td>
<td><a href="http://www.ontotext.com/example/wine#madeFromGrape">http://www.ontotext.com/example/wine#madeFromGrape</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/2000/01/rdf-schema#label">http://www.w3.org/2000/01/rdf-schema#label</a></td>
</tr>
<tr>
<td>Default value</td>
<td>default value</td>
</tr>
<tr>
<td>Datatype</td>
<td></td>
</tr>
<tr>
<td>Indexed</td>
<td>✔</td>
</tr>
<tr>
<td>Stored</td>
<td>✔</td>
</tr>
<tr>
<td>Analyzed</td>
<td>✔</td>
</tr>
<tr>
<td>Multivalued</td>
<td>✔</td>
</tr>
<tr>
<td>Facet</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field name</th>
<th>sugar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property chain</td>
<td><a href="http://www.ontotext.com/example/wine#hasSugar">http://www.ontotext.com/example/wine#hasSugar</a></td>
</tr>
<tr>
<td>Default value</td>
<td>default value</td>
</tr>
<tr>
<td>Datatype</td>
<td></td>
</tr>
<tr>
<td>Indexed</td>
<td>✔</td>
</tr>
<tr>
<td>Stored</td>
<td>✔</td>
</tr>
<tr>
<td>Analyzed</td>
<td>✔</td>
</tr>
<tr>
<td>Multivalued</td>
<td>✔</td>
</tr>
<tr>
<td>Facet</td>
<td>✔</td>
</tr>
</tbody>
</table>

#### Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>language (e.g. en, bg)</th>
</tr>
</thead>
</table>

#### Types

<table>
<thead>
<tr>
<th>Type</th>
<th><a href="http://www.ontotext.com/example/wine#Wine">http://www.ontotext.com/example/wine#Wine</a></th>
</tr>
</thead>
</table>

#### Entity filter

<table>
<thead>
<tr>
<th>Filter</th>
<th>?a in (&quot;value&quot;, &quot;other value&quot;) and bound(?b)</th>
</tr>
</thead>
</table>

#### Boost properties

<table>
<thead>
<tr>
<th>Property</th>
<th>URI</th>
</tr>
</thead>
</table>

#### Strip markup

<table>
<thead>
<tr>
<th>Strip markup</th>
<th></th>
</tr>
</thead>
</table>

#### Analyzer

<table>
<thead>
<tr>
<th>Analyzer</th>
<th>Lucene analyzer, i.e. org.apache.lucene.analysis.en.EnglishAnalyzer</th>
</tr>
</thead>
</table>

---

**View SPARQL Query**

[Cancel] [OK]
Using the create command

The create command is triggered by a SPARQL `INSERT` with the `createConnector` predicate, e.g., it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above:

```sql
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.ontotext.com/example/wine#Wine"
    ],
    "fields": [
      {
        "fieldName": "grape",
        "propertyChain": [
          "http://www.ontotext.com/example/wine#madeFromGrape",
          "http://www.w3.org/2000/01/rdf-schema#label"
        ],
        "analyzed": false,
        "multivalued": false
      },
      {
        "fieldName": "sugar",
        "propertyChain": [
          "http://www.ontotext.com/example/wine#hasSugar"
        ],
        "analyzed": false,
        "multivalued": false
      },
      {
        "fieldName": "year",
        "propertyChain": [
          "http://www.ontotext.com/example/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 and, in the example, it is only entities of the type `http://www.ontotext.com/example/wine#Wine` (and its subtypes). The "fields" key defines the mapping from RDF to Lucene. The basic building block is the property chain, i.e., 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 later used in the queries.

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. 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`, `2013`, and thus it is best to specify the option `analyzed: false` as well. See `analyzed` in Defining fields for more information.
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 \texttt{INSERT} with the \texttt{dropConnector} predicate where the name of the connector instance has to be in the subject position, e.g., this removes the connector \texttt{my\_index}:

\begin{verbatim}
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 [] .
}
\end{verbatim}

You can also force drop a connector in case a normal delete 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, and check Force delete in the dialog box.

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:

\begin{verbatim}
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 .
}
\end{verbatim}

Listing available connector instances

In the Connectors management view

Existing Connector instances show under Existing connectors (below the New Connector button). Click the name of an instance to view its configuration and SPARQL query, or click the repair / delete icons to perform these operations.
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 .
}
```

?cntUri is bound to the prefixed IRI of the connector instance that was used during creation, e.g., `http://www.ontotext.com/connectors/lucene/instance#my_index`, while ?cntStr is bound to a string, representing the part after the prefix, e.g., "my_index".
Instance status check

The internal state of each connector instance can be queried using a SELECT query and the `connectorStatus` predicate:

```sparql
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
SELECT ?cntUri ?cntStatus {
}
```

`?cntUri` is bound to the prefixed 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.

5.14.2.4 Working with data

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, i.e., you must simply execute standard SPARQL INSERT/DELETE queries. This is achieved by intercepting all changes in the plugin and determining which abstract documents need to be updated.

Simple queries

Once a connector instance has been created, it is possible to query data from it through SPARQL. For each matching abstract document, the connector instance returns the document subject. In its simplest form, querying is achieved by using a SELECT and providing the Lucene query as the object of the `query` predicate:

```sparql
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 binds `?entity` to the two wines made from grapes that have “cabernet” in their name, namely :Yowowine 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.

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 :query.
3. Request the matching entities through the :entities predicate.

It is also possible to provide per query search options by using one or more option predicates. The option predicates are described in detail below.
Combining Lucene results with GraphDB data

The bound ?entity can be used in other SPARQL triples in order to build complex queries that fetch additional data from GraphDB, for example, to see the actual grapes in the matching wines as well as the year they were made:

```sparql
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX wine: <http://www.ontotext.com/example/wine#>

SELECT ?entity ?grape ?year {
  ?search a luc-index:my_index ;
    luc:query "grape:cabernet" ;
    luc:entities ?entity .
  ?entity wine:madeFromGrape ?grape .
  ?entity wine:hasYear ?year
}
```

The result looks like this:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?grape</th>
<th>?year</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>:CabernetSauvignon</td>
<td>2013</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:Merlo</td>
<td>2012</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:CabernetFranc</td>
<td>2012</td>
</tr>
</tbody>
</table>

**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 Lucene with the `score` predicate. As each entity has its own score, the predicate should come at the entity level. For example:

```sparql
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:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?score</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>0.9442668212516785</td>
</tr>
<tr>
<td>:Franvino</td>
<td>0.7554128170013428</td>
</tr>
</tbody>
</table>
Basic facet queries

Consider the sample wine data and the my_index connector instance described previously. You can also query facets using the same instance:

```
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 _:f .
  _:f luc:facetName ?facetName .
  _:f luc:facetValue ?facetValue .
  _:f 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 facets predicate. As each facet has three components (name, value and count), the facets predicate binds a blank node, which in turn 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:

<table>
<thead>
<tr>
<th>facetName</th>
<th>facetValue</th>
<th>facetCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>2012</td>
<td>3</td>
</tr>
<tr>
<td>year</td>
<td>2013</td>
<td>2</td>
</tr>
<tr>
<td>sugar</td>
<td>dry</td>
<td>3</td>
</tr>
<tr>
<td>sugar</td>
<td>medium</td>
<td>2</td>
</tr>
</tbody>
</table>

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.

Sorting

It is possible to sort the entities returned by a connector query according to one or more fields. Sorting is achieved by the orderBy predicate the value of which is a comma-delimited list of fields. 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#>

SELECT ?entity {
  ?search a luc-index:my_index ;
  luc:query "year:2013" ;
  luc:orderBy "-sugar" ;
  luc:entities ?entity .
}
```

The result contains wines produced in 2013 sorted according to their sugar content in descending order:

- entity
  - Rozova
  - Yoyowine

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. To remedy this, use `ORDER BY` from SPARQL.

**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:** Unlike Lucene 4, which was used in GraphDB 6.x, Lucene 5 imposes an additional requirement on fields used for sorting. They must be defined with `multivalued = false`.

### Limit and offset

Limit and offset are supported on the Lucene side of the query. This is achieved through the predicates `limit` and `offset`. Consider this example in which an offset of 1 and a limit of 1 are specified:

```sparql
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 .
}
```

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
Yoyowine
Franvino
Blanquito
```

**Note:** The specific order in which GraphDB returns the results depends on how Lucene returns the matches, unless sorting is specified.

### Snippet extraction

Snippet extraction is used for extracting highlighted snippets of text that match the query. The snippets are accessed through the dedicated predicate `snippets`. It binds a blank node that in turn provides the actual snippets via the predicates `snippetField` and `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:

```sparql
PREFIX luc: <http://www.ontotext.com/connectors/lucene#>
PREFIX luc-index: <http://www.ontotext.com/connectors/lucene/instance#>
```

(continues on next page)
the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

<table>
<thead>
<tr>
<th>entity</th>
<th>snippetField</th>
<th>snippetText</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yoyowine</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt; Sauvignon</td>
</tr>
<tr>
<td>Franvino</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt; Franc</td>
</tr>
</tbody>
</table>

**Note:** The actual snippets might be different as this depends on the specific Lucene implementation.

It is possible to tweak how the snippets are collected/composed by using the following option predicates:

- `:snippetSize` - sets the maximum size of the extracted text fragment, 250 by default;
- `:snippetSpanOpen` - text to insert before the highlighted text, `<em>` by default;
- `:snippetSpanClose` - text to insert after the highlighted text, `</em>` by default.

The option predicates are set on the query instance, much like the `:query` predicate.

**Total hits**

You can get the total number of hits by using the `totalHits` predicate, e.g., for the connector instance `my_index` and a query that retrieves all wines made in 2012:

```
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`.

**5.14.2.5 List of creation parameters**

The creation parameters define how a connector instance is created by the `:createConnector` predicate. 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 Lucene index for tracking updates to chains)

The only way to index changes in data after the connector has been created is to repair (or drop/recreate) the connector.

**importGraph (boolean), optional, a virtual graph containing data from which to create the connector**

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: <http://www.ontotext.com/connectors/lucene#>` control prefix is used. Data needs to be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and create statement must be in the same transaction. In GDB Workbench, this can be done by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This functionality requires `readonly mode`.

```sql
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 functionality requires `readonly mode`.

**detectFields (boolean), optional, detects fields**  This mode introduces automatic fields detection when creating a connector. Instead of providing fields: [ ... ] in the JSON, you can skip this and get automatic fields, where each automatic field will have a single predicate chain and its field name will the same as the predicate.

In this mode, specifying types is optional too. 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 the Workbench.

**analyzer (string), optional, specifies Lucene analyzer**  The Lucene Connector supports custom Analyzer implementations. They may be specified via 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:

```java
package com.ontotext.example;

import org.apache.lucene.analysis.Analyzer;

public class FancyAnalyzer extends Analyzer {

(continues on next page)
FancyAnalyzer and SmartAnalyzer can then be used by specifying their fully qualified names, for example:

```java
... "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. This can be done 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. In addition, an empty range can be used 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 to keep the field names simple.

- **fieldNameTransform (one of none, predicate or predicate.localName), optional, none by default** The field name transform defines an optional transformation of the field name.
  - `none`: The field name is supplied via the `fieldName` option.
  - `predicate`: The field name is derived from the full IRI of the last predicate of the chain, e.g., if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will be `http://www.w3.org/2000/01/rdf-schema#label` too.
  - `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain, e.g., if the last predicate was `http://www.w3.org/2000/01/rdf-schema#comment`, then the field name will be `comment`. 

See also the examples in General full-text search with the connectors.
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 defined as 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. The default value (defaultValue) provides means for specifying 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. Fields can be stored in Lucene and this is controlled by the Boolean option “stored”. Stored fields are required for retrieving snippets. 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 (they 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 that 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. Lucene needs to index data in a special way, if it will be used for faceted search. 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**.

The datatype mapping can be overridden through the parameter "datatype", 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**.

**Special field definitions**

**Copy fields**

Often, it is convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases, e.g., faceting or sorting vs full-text search. The Lucene GraphDB Connector has explicit support for fields that copy their value from another field. This is achieved by specifying a single element in the property chain of the form @otherFieldName, where otherFieldName is another non-copy field. Take the following example:

```json
"fields": [
  {
    "fieldName": "grape",
    "propertyChain": [
      "http://www.ontotext.com/example/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 “grape”.

**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, e.g., 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 alphanumeric sequence of convenience. For example, we can define the fields name$1$ and name$2$ like this:

```json
...
"fields": [
  {
    "fieldName": "name$1",
    "propertyChain": [
      "http://www.ontotext.com/example#canonicalName"
    ],
    "fieldName": "name$2",
    "propertyChain": [
      "http://www.ontotext.com/example#historicalName"
    ]
  },
  ...
]
```

The values of the fields name$1$ and name$2$ will be merged and synchronized to the field name in Lucene.

**Note:** You cannot mix suffixed and unsuffixed fields with the same name, e.g., if you defined myField$new$ and myField$old$ you cannot have a field called just myField.

Filters and fields with multiple property chains

Filters can be used 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, e.g., ?myField
- Virtual fields are specified with the suffix, e.g., ?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, where the last element is the pseudo-IRI lang(). The property preceding lang() must lead to a literal value. For example:

```sparql
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
}
```

(continues on next page)
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

The named graph of a given value can be indexed 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.

```json
INSERT DATA {
  luc-index:my_index luc:createConnector 
  {
    "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 named graph of each value of the property http://www.ontotext.com/example#name into the field nameGraph.
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:

```json
{
    "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 (e.g., [http://www.ontotext.com/example/wine#Fravino](http://www.ontotext.com/example/wine#Fravino) from our small example dataset) indexed as a regular field. This can be achieved by specifying a property chain with a single property referring to the pseudo-IRI $self. For example:

```prefix
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.ontotext.com/example/wine#Wine" ],
     "fields": [ {
      "fieldName": "entityId",
      "propertyChain": [ "$self" ]}, {
      "fieldName": "grape",
      "propertyChain": [ "http://www.ontotext.com/example/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.
5.14.2.6 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:

<table>
<thead>
<tr>
<th>RDF value</th>
<th>RDF datatype</th>
<th>Indexed in Lucene as</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRI</td>
<td>n/a</td>
<td>Field (not tokenized)</td>
</tr>
<tr>
<td>literal</td>
<td>any type not explicitly mentioned below</td>
<td>Field (tokenized, with term vectors)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:boolean</td>
<td>Field (not tokenized), with the values true and false</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:double</td>
<td>DoublePoint</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:float</td>
<td>FloatPoint</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:long</td>
<td>LongPoint</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:int</td>
<td>LongPoint</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:dateTime</td>
<td>Field (not tokenized), padded string with second precision</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:date</td>
<td>Field (not tokenized), padded string with day precision</td>
</tr>
</tbody>
</table>

The datatype mapping can be affected by the synchronization options too, e.g., 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, e.g., the first value has no datatype but other values have.

It is therefore recommended to set datatype to a fixed value, e.g. xsd:date.

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 strings for range search, e.g., "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, i.e., 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 as strings.

XSD date and time values support timezones. In order to have a unified view over values with different timezones, all date and time 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, we define the undetermined time zone as UTC, i.e., "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. E.g., "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 strict timezone adherence is followed. We have chosen to ignore the timezone on any values that do not have an associated time value, e.g.:

- "2020-02-15+02:00"^^xsd:date
- "2020-05-08-05:00"^^xsd:date

All of the above will be treated as if they specified UTC as their timezone.

### 5.14.2.7 Entity filtering

The Lucene connector supports three kinds of entity filters used to fine-tune the set of entities and/or 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 `FILTER()` inside a SPARQL query but not exactly the same. In them, each configured field can be referred to by prefixing it with a `?`, much like referring to a variable in SPARQL.

#### Types of filters

**Top-level value filter** The top-level value filter is specified via `valueFilter`. 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 and 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 via `documentFilter`. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. It can be used to enforce global document restrictions, e.g., 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 via `valueFilter` 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 `$this`. Other field names can be used to filter the current field’s value based on the value of another field, e.g., `$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.*
Filter operators

The filter operators are used to test if the value of a given field satisfies some condition:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| `?var in (value1, value2, ...)` | Tests if the field `var`’s value is one of the specified values. Values are compared strictly unlike the similar SPARQL operator, i.e. for literals to match their datatype must be exactly the same (similar to how SPARQL `sameTerm` works). 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 field `var`’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 4 – continued from previous page

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?var = value (equal to)</td>
<td>RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The field var’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.</td>
</tr>
<tr>
<td>?var != value (not equal to)</td>
<td></td>
</tr>
<tr>
<td>?var &gt; value (greater than)</td>
<td>Example: Given that height’s value is &quot;150&quot;^^xsd:int and dateOfBirth’s value is &quot;1989-12-31&quot;^^xsd:date, then:</td>
</tr>
<tr>
<td>?var &gt;= value (greater than or equal to)</td>
<td></td>
</tr>
<tr>
<td>?var &lt; value (less than)</td>
<td></td>
</tr>
<tr>
<td>?var &lt;= value (less than or equal to)</td>
<td></td>
</tr>
<tr>
<td>regex(?var, &quot;pattern&quot;)</td>
<td>Tests if the field var’s value matches the given regular expression pattern. If the “i” flag option is present, this indicates that the match operates in case-insensitive mode. Values that do not match are treated as if they were not present in the repository.</td>
</tr>
<tr>
<td>regex(?var, &quot;pattern&quot;, &quot;i&quot;)</td>
<td>Example:</td>
</tr>
<tr>
<td>expr1</td>
<td></td>
</tr>
<tr>
<td>expr1 or expr2</td>
<td>Examples:</td>
</tr>
<tr>
<td>expr1 &amp; expr2</td>
<td>Logical conjunction of expressions expr1 and expr2.</td>
</tr>
<tr>
<td>expr1 and expr2</td>
<td>Examples:</td>
</tr>
<tr>
<td>!expr</td>
<td>Logical negation of expression expr.</td>
</tr>
</tbody>
</table>

Continued on next page
### Filter modifiers

In addition to the operators, there are some constructions that can be used to 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)` is used for going to a previous level in a property chain. It can be applied recursively as many times as needed, e.g., \( \text{parent(parent(parent(?var)))} \) goes back in the chain three times. The effective value of `parent(?var)` can be used with the `in` or `not in` operator like this: `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator like this: `parent(bound(?var))`.

**Accessing an element beyond the chain** The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) is used to access additional values that are accessible through the property `uri`. In essence, this construction corresponds to the triple pattern \( \text{value uri ?effectiveValue} \), where `?value` is a value bound by the field `var`. The effective value of `?var -> uri` can be used with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. It can be combined with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. The same construction can be applied to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or even combined with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The `IRI` parameter can be a full IRI within `< >` or the special string `rdf:type` (alternatively, 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)` is used for accessing 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(?x)` is the recommended way.

The construction can be combined with `parent()` like this: `graph(parent(?a)) in (<urn:a>)`.

**Filtering by language tags** The construction `lang(?var)` is used for accessing 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 ('en', 'bg')`. The construction can be combined with `parent()` and an element beyond the chain like this: `lang(parent(?a) -> http://www.w3.org/2000/01/rdf-schema#label) in ('en', 'bg')`. Literal values without language tags can be filtered by using an empty tag: `''`.

**Current context variable $this** The special field variable `$this` (and not `?this`, `$?this`, `??this`) is used to refer to the current context. In the top-level value filter and the top-level document filter it refers to the document ID. In the per-field value filter it refers to the currently filtered field value.

**ALL() quantifier** In the context of document-level filtering, a match is true if at least one of potentially many field values match, e.g., `?location = <urn:Europe>` would return true if the document contains \{ "location": ["<urn:Asia>", "<urn:Europe>"] \}. In addition to this, you can also use the `ALL()` quantifier when you need all values to match, e.g., `ALL( ?location ) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

**Entity filters and default values** Entity filters can be combined with default values in order to get more flexible behavior.

A typical use-case for an entity filter is having soft deletes, i.e., instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

---

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>( expr )</code></td>
<td>Grouping of expressions</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td><code>(bound(?name) or bound(?company)) &amp;&amp; bound(?address)</code></td>
</tr>
</tbody>
</table>
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 comparing the field value to 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 and the top-level 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 as 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 being indexed first.

Basic entity filter example

Given the following RDF data:

```xml
@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:

```ruby
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.ontotext.com/example#gadget"],
    "fields": [
      {
        "fieldName": "name",
        "propertyChain": ["http://www.ontotext.com/example#name"]
      }
    ]
  }
}
```

(continues on next page)
The entity :beta is not synchronized as 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"
  },
  "documentFilter": "bound(?city)"
}
```

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.

**Advanced entity filter example**

Sometimes, data represented in RDF is not well suited to map directly to non-RDF. For example, if you have news articles and they can be tagged with different concepts (locations, persons, events, etc.), one possible way to model this is a single property :taggedWith. Consider the following RDF data:

```html
@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
  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 .

Now, if you map this data to Lucene so that the property :taggedWith is mapped to separate fields taggedWithPerson and taggedWithLocation according to the type of x (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.ontotext.com/example2#Article"],
     "fields": [
       { "fieldName": "comment",
         "propertyChain": ["http://www.w3.org/2008/01/rdf-schema#comment"]
       },
       { "fieldName": "taggedWithPerson",
         "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
         "valueFilter": "$this -> type in (<http://www.ontotext.com/example2#Person>)"
       },
       { "fieldName": "taggedWithLocation",
         "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
         "valueFilter": "$this -> type in (<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:
This can be checked by issuing a faceted search for `taggedWithLocation` and `taggedWithPerson`:

```sparql
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 _:f ;
  _:f 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`:

<table>
<thead>
<tr>
<th>?facetName</th>
<th>?facetValue</th>
<th>?facetCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>taggedWithLocation</td>
<td><a href="http://www.ontotext.com/example2#Berlin">http://www.ontotext.com/example2#Berlin</a></td>
<td>3</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Mozart">http://www.ontotext.com/example2#Mozart</a></td>
<td>2</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Einstein">http://www.ontotext.com/example2#Einstein</a></td>
<td>1</td>
</tr>
</tbody>
</table>

### 5.14.2.8 Overview of connector predicates

The following diagram shows a summary of all predicates that can administrate (create, drop, check status) 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.
5.14.2.9 Caveats

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 need to come before any predicates that fetch results.

The diagram in *Overview of connector predicates* provides a quick overview of the predicates.
5.14.2.10 Upgrading from previous versions

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 \texttt{BOUND()} \rightarrow rewrite with per-field value filter.
   - If you want to remove entire documents, i.e., if the operand is \texttt{BOUND()} \rightarrow rewrite with top-level document filter.

   So if we take the example:

   \begin{verbatim}
   ?location = <urn:Europe> AND \texttt{BOUND(?location)} AND ?type IN (<urn:Foo>, <urn:Bar>)
   \end{verbatim}

   It needs to be rewritten like this:
   - Per-field rule on field \texttt{location: }$this = <urn:Europe>$
   - Per-field rule on field \texttt{type: }$this$ \texttt{IN (} <urn:Foo>, <urn:Bar> \texttt{)}
   - Top-level document filter: \texttt{BOUND(?location)}
4. Recreate the connector instance using the new definition.

5.14.3 Solr GraphDB connector

\textbf{Note:} This feature requires a GraphDB Enterprise license.

5.14.3.1 Overview and features

The GraphDB Connectors provide extremely fast normal and faceted (aggregation) searches, typically implemented by an external component or a service such as Solr but have 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 (a 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 defined as a sequence of triples where each triple’s object is the subject of the following triple.

The main features of the GraphDB Connectors are:

- maintaining an index that is always in sync with the data stored in GraphDB;
- multiple independent instances per repository;
- the entities for synchronization are 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`'s 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.

5.14.3.2 Usage

All interactions with the Solr GraphDB Connector shall be done through SPARQL queries.

There are three types of SPARQL queries:
– `INSERT` for creating and deleting connector instances;
– `SELECT` for listing connector instances and querying their configuration parameters;
– `INSERT/SELECT` for storing and querying data as part of the normal GraphDB data workflow.

In general, this corresponds to `INSERT` adds or modifies data and `SELECT` queries existing data.

Each connector implementation defines its own IRI prefix 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 this prefix, e.g., `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 prefix to avoid clashing with any of the command predicates. For Solr, the instance prefix is `http://www.ontotext.com/connectors/solr/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.ontotext.com/example/wine#> .

wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlo
  rdf:type wine:Grape ;
  rdfs:label "Merlo" .

wine:CabernetSauvignon
  rdf:type wine:Grape ;
  rdfs:label "Cabernet Sauvignon" .
```

(continues on next page)
5.14.3.3 Setup and maintenance

Prerequisites

**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` folder of the GraphDB distribution to your Solr home.

2. Tell Solr to scan the `.jar` and use the GraphDB custom handler instead of the default one. Add it to the root `solr` tag in `solr.xml` in your Solr home:

```xml
<str name="adminHandler">com.ontotext.solr.handler.admin.GraphDBConnectorAdminHandler</str>
<str name="sharedLib">${sharedLib:}</str>
```

To start Solr with custom `solr-home`:

```bash
solr start
```
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  This version of the Solr GraphDB Connector uses Solr version 8.11.1.

Creating a connector instance

Creating a connector instance is done by sending a SPARQL query with the following configuration data:

- the name of the connector instance (e.g., my\_index);
- a Solr instance to synchronize to;
- classes to synchronize;
- properties to synchronize.

The configuration data has to be provided as a JSON string representation and passed together with the create command.

If you create the connector via the Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using the 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.
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 `createConnector` predicate, e.g., it creates a connector instance called `my_index`, which synchronizes the wines from the sample data above:

```sparql
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/wine#Wine"
    ],
    "fields": [
      {
        "fieldName": "grape",
        "propertyChain": [
          "http://www.ontotext.com/example/wine#madeFromGrape",
          "http://www.w3.org/2000/01/rdf-schema#label"
        ],
        "analyzed": false,
        "multivalued": false
      },
      {
        "fieldName": "sugar",
        "propertyChain": [
          "http://www.ontotext.com/example/wine#hasSugar"
        ],
        "analyzed": false,
        "multivalued": false
      },
      {
        "fieldName": "year",
        "propertyChain": [
          "http://www.ontotext.com/example/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 and, in the example, it is only entities of the type `http://www.ontotext.com/example/wine#Wine` (and its subtypes). The "fields" key defines the mapping from RDF to Solr. The basic building block is the property chain, i.e., 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 later used in the queries.

Grape is an example of a property chain composed of more than one property. First, we take the wine’s made-FromGrape property, the object of which is an instance of the type Grape, and then we take the rdfs:label of this instance. 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, 2013, and thus 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:

```xml
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/index#>

INSERT DATA {
  { solr-index:my_index solr:createConnector "";  
   "solrUrl": "http://localhost:8983/solr",  
   "types": [  
     "http://www.ontotext.com/example/wine#Wine"  
   ],  
   "fields": [  
     {  
       "fieldName": "grape",  
       "propertyChain": [  
         "http://www.ontotext.com/example/wine#madeFromGrape",  
         "http://www.w3.org/2000/01/rdf-schema#label"  
       ]  
     },  
     {  
       "fieldName": "sugar",  
       "propertyChain": [  
         "http://www.ontotext.com/example/wine#hasSugar"  
       ],  
       "analyzed": false,  
       "multiValued": false  
     },  
     {  
       "fieldName": "year",  
       "propertyChain": [  
         "http://www.ontotext.com/example/wine#hasYear"  
       ],  
       "analyzed": false  
     }  
   ],  
   "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:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Solr config</th>
</tr>
</thead>
<tbody>
<tr>
<td>_graphdb_id</td>
<td><code>&lt;field name=&quot;_graphdb_id&quot; type=&quot;string&quot; indexed=&quot;true&quot; stored=&quot;true&quot; required=&quot;true&quot; multiValued=&quot;false&quot;/&gt;</code></td>
</tr>
<tr>
<td>grape</td>
<td><code>&lt;field name=&quot;grape&quot; type=&quot;text_general&quot; indexed=&quot;true&quot; stored=&quot;true&quot; multiValued=&quot;true&quot;/&gt;</code></td>
</tr>
<tr>
<td>sugar</td>
<td><code>&lt;field name=&quot;sugar&quot; type=&quot;text_general&quot; indexed=&quot;true&quot; stored=&quot;true&quot; multiValued=&quot;false&quot;/&gt;</code></td>
</tr>
<tr>
<td>year</td>
<td><code>&lt;field name=&quot;year&quot; type=&quot;tints&quot; indexed=&quot;true&quot; stored=&quot;true&quot; multiValued=&quot;true&quot;/&gt;</code></td>
</tr>
</tbody>
</table>

_*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.

**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 *dropConnector* predicate where the name of the connector instance has to be in the subject position, e.g., this 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 in case a normal delete 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, and check Force delete in the dialog box.

![Confirm delete dialog box]

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:

```sql
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 .
}
```

Listing available connector instances

In the Connectors management view

Existing Connector instances show under Existing connectors (below the New Connector button). Click the name of an instance to view its configuration and SPARQL query, or click the repair / delete icons to perform these operations.
With a SPARQL query

Listing connector instances returns all previously created instances. It is a SELECT query with the `listConnectors` predicate:

```sparql
PREFIX solr: <http://www.ontotext.com/connectors/solr#>

SELECT ?cntUri ?cntStr {
  ?cntUri solr:listConnectors ?cntStr .
}
```

`?cntUri` is bound to the prefixed IRI of the connector instance that was used during creation, e.g., `http://www.ontotext.com/connectors/solr/instance#my_index`, while `?cntStr` is bound to a string, representing the part after the prefix, e.g., "my_index".
Instance status check

The internal state of each connector instance can be queried using a SELECT query and the `connectorStatus` predicate:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
SELECT ?cntUri ?cntStatus {
}
```

`?cntUri` is bound to the prefixed 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.

5.14.3.4 Working with data

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, i.e., you must simply execute standard SPARQL `INSERT/DELETE` queries. This is achieved by intercepting all changes in the plugin and determining which abstract documents need to be updated.

Simple queries

Once a connector instance has been created, it is possible to query data from it through SPARQL. For each matching abstract document, the connector instance returns the document subject. In its simplest form, querying is achieved by using a SELECT and providing the Solr query as the object of the `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 binds `?entity` to the two wines made from grapes that have “cabernet” in their name, namely :Yowowine 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.

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 :query.
3. Request the matching entities through the :entities predicate.

It is also possible to provide per query search options by using one or more option predicates. The option predicates are described in detail below.
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:

```sparql
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",
            "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:

```sparql
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=*&%A&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

The bound ?entity can be used in other SPARQL triples in order to build complex queries that fetch additional data from GraphDB, for example, to see the actual grapes in the matching wines as well as the year they were made:

```sparql
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>
PREFIX wine: <http://www.ontotext.com/example/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:
<table>
<thead>
<tr>
<th>?entity</th>
<th>?grape</th>
<th>?year</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>:CabernetSauvignon</td>
<td>2013</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:Merlot</td>
<td>2012</td>
</tr>
<tr>
<td>:Franvino</td>
<td>:CabernetFranc</td>
<td>2012</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?score</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>0.9442660212516785</td>
</tr>
<tr>
<td>:Franvino</td>
<td>0.7554128170013428</td>
</tr>
</tbody>
</table>

### Basic facet queries

Consider the sample wine data and the `my_index` connector instance described previously. You can also query facets using the same instance:

```
PREFIX solr: <http://www.ontotext.com/connectors/solr#>
PREFIX solr-index: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?facetAddress ?facetAddressValue ?facetAddressCount WHERE {
  # note empty query is allowed and will just match all documents, hence no :query
  ?r a solr-index:my_index ;
  solr:facetAddressFields "year,sugar" ;
  solr:facetcount _:_ .
  _f solr:facetAddress ?facetAddress .
  _f solr:facetAddressValue ?facetAddressValue .
  _f solr:facetAddressCount ?facetAddressCount .
}
```

It is important to specify the facet fields by using the `facetAddressFields` predicate. Its value is a simple comma-delimited list of field names. In order to get the faceted results, use the facets predicate. As each facet has three components (name, value and count), the facets predicate binds a blank node, which in turn can be used to access the individual values for each component through the predicates `facetAddressName`, `facetAddressValue`, and `facetAddressCount`. The resulting bindings look like the following:
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 analysed textual field works but might produce unexpected results. Analysed 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](#).

### 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:

<table>
<thead>
<tr>
<th>predicate</th>
<th>meaning</th>
<th>Solr counterpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>:name</td>
<td>Bucket name</td>
<td>getName()</td>
</tr>
<tr>
<td>:key</td>
<td>Key or value associated with the bucket</td>
<td>getValue() or getKey()</td>
</tr>
<tr>
<td>:count</td>
<td>Count of documents in the bucket</td>
<td>getCount()</td>
</tr>
<tr>
<td>:from</td>
<td>Start of range (RangeFacet)</td>
<td>getStart()</td>
</tr>
<tr>
<td>:to</td>
<td>End of range (RangeFacet)</td>
<td>getEnd()</td>
</tr>
<tr>
<td>:rangeGap</td>
<td>Gap of range (RangeFacet)</td>
<td>getGap()</td>
</tr>
<tr>
<td>:beforeCount</td>
<td>Count of documents before the first range</td>
<td>getBefore()</td>
</tr>
<tr>
<td>:afterCount</td>
<td>Count of documents after the first range</td>
<td>getAfter()</td>
</tr>
<tr>
<td>:betweenCount</td>
<td>Count of documents within all ranges</td>
<td>getBetween()</td>
</tr>
<tr>
<td>:parent</td>
<td>Pivot facets: points to the parent (upper level) blank node</td>
<td></td>
</tr>
<tr>
<td>:level</td>
<td>Pivot facets: level number where 1 is the uppermost level and the following levels are 2, 3 and so on</td>
<td>getLevel()</td>
</tr>
<tr>
<td>:levelName</td>
<td>Pivot facets: level name</td>
<td>getField()</td>
</tr>
</tbody>
</table>
Sorting

It is possible to sort the entities returned by a connector query according to one or more fields. Sorting is achieved by the `orderBy` predicate the value of which is a comma-delimited list of fields. 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#>

SELECT ?entity {
?search a solr-index:my_index ;
  solr:query "year:2013" ;
  solr:orderBy "-sugar" ;
  solr:entities ?entity .
}
```

The result contains wines produced in 2013 sorted according to their sugar content in descending order:

entity
Rozova
Yoyowine

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. To remedy this, use `ORDER BY` from SPARQL.

**Tip:** Sorting by an analysed textual field works but might produce unexpected results. Analysed 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](Copy%20fields).

**Note:** Solr imposes an additional requirement on fields used for sorting. They must be defined with `multivalued = false`.

Limit and offset

Limit and offset are supported on the Solr side of the query. This is achieved through 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 .
}
```

The result contains a single wine, Franvino. If you execute the query without the limit and offset, Franvino will be second in the list:
**Note:** The specific order in which GraphDB returns the results depends on how Solr returns the matches, unless sorting is specified.

### Snippet extraction

Snippet extraction is used for extracting highlighted snippets of text that match the query. The snippets are accessed through the dedicated predicate `snippets`. It binds a blank node that in turn provides the actual snippets via the predicates `snippetField` and `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:

```sparql
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 .
  ?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:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?snippetField</th>
<th>?snippetText</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Yoyowine</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt;/ Sauvignon</td>
</tr>
<tr>
<td>:Franvino</td>
<td>grape</td>
<td>&lt;em&gt;Cabernet&lt;/em&gt;/ Franc</td>
</tr>
</tbody>
</table>

**Note:** The actual snippets might be different as this depends on the specific Solr implementation.

It is possible to tweak how the snippets are collected/composed by using the following option predicates:

- `:snippetSize` - sets the maximum size of the extracted text fragment, 250 by default;
- `:snippetSpanOpen` - text to insert before the highlighted text, `<em>` by default;
- `:snippetSpanClose` - text to insert after the highlighted text, `</em>` by default.

The option predicates are set on the query instance, much like the `:query` predicate.
Total hits

You can get the total number of hits by using the `totalHits` predicate, e.g., for the connector instance `my_index` and a query that retrieves all wines made in 2012:

```sparql
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 `xsd:long`) binds to `?totalHits`.

5.14.3.5 List of creation parameters

The creation parameters define how a connector instance is created by the `:createConnector` predicate. 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 Lucene index for tracking updates to chains)

The only way to index changes in data after the connector has been created is to repair (or drop/recreate) the connector.

**importGraph** (boolean), optional, a virtual graph containing data from which to create the connector

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 PREFIIX `solr`: <http://www.ontotext.com/connectors/solr#> control prefix is used. Data needs to be inserted into this graph before the connector create statement is executed.

Both the insertion into the special graph and create statement must be in the same transaction. In GDB Workbench, this can be done by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This functionality requires `readonly mode`.

```sparql
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 ""
} {
  "readonly": true,
  "importGraph": true,
  "fields": []
}
```
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 functionality requires `readonly mode`.

detectFields (boolean), optional, detect fields

This mode introduces automatic fields detection when creating a connector. Instead of providing fields: `[ ... ]` in the JSON, you can skip this and get automatic fields, where each automatic field will have a single predicate chain and its field name will the same as the predicate.

In this mode, specifying types is optional too. 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 the 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 format of the URL is of the form `http://hostname.domain:port/`. There is no default value. Can be updated at runtime without having to rebuild the index.

solrBasicAuthUser (string), optional, the settings for supplying the authentication user

No default value. Can be updated at runtime without having 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 without having 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 without having 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. This can be done 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. In addition, an empty range can be used 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 to keep the field names simple.

- **fieldNameTransform (one of none, predicate or predicate.localName), optional, none by default**
  The field name transform defines an optional transformation of the field name.
  - none: The field name is supplied via the `fieldName` option.
– **predicate**: The field name is derived from the full IRI of the last predicate of the chain, e.g., if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will be `http://www.w3.org/2000/01/rdf-schema#label` too.

– **predicate.localName**: The field name is derived from the local name of the IRI of the last predicate of the chain, e.g., 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 defined as 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 The default value (`defaultValue`) provides means for specifying 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 option corresponds to the property “indexed” in the Solr schema.

**stored** (boolean), optional, default `true` Fields can be stored in Solr and this is controlled by the Boolean option “stored”. Stored fields are required for retrieving snippets. `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 analysed according to the analyser settings. Should you require that a given field is not analysed, you may use “analyzed”. This option has no effect for IRIs (they are never analysed). `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 analysed 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 that 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, seeDatatype mapping.

The mapping can be overridden through the property “datatype”, 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.

### Updating parameters at runtime

As mentioned above, the following connector parameters can be updated at runtime without having to rebuild the index:

- solrUrl
- bulkUpdateBatchSize
- solrBasicAuthUser
- solrBasicAuthPassword

This can be done by executing the following SPARQL update, here with examples for changing the user and password:

```sparql
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"
  }
  ...'
}
```

### Special field definitions

#### Copy fields

Often, it is convenient to synchronize one and the same data multiple times with different settings to accommodate for different use cases, e.g., faceting or sorting vs full-text search. The Solr GraphDB Connector has explicit support for fields that copy their value from another field. This is achieved by specifying a single element in the property chain of the form @otherFieldName, where otherFieldName is another non-copy field. Take the following example:

```json
...
"fields": {

(continues on next page)
```
The snippet creates an analysed field “grape” and a non-analysed 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, e.g., 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 alphanumeric sequence of convenience. For example, we can define the fields `name$1` and `name$2` like this:

```json
... "fields": [  
    {  
        "fieldName": "name$1",  
        "propertyChain": [  
          "http://www.ontotext.com/example#canonicalName"  
        ],  
        "fieldName": "name$2",  
        "propertyChain": [  
          "http://www.ontotext.com/example#historicalName"  
        ]  
    }  
... ]
```

The values of the fields `name$1` and `name$2` will be merged and synchronized to the field `name` in Solr.

**Note:** You cannot mix suffixed and unsuffixed fields with the same same, e.g., if you defined `myField$new` and `myField$old` you cannot have a field called just `myField`. 
Filters and fields with multiple property chains

Filters can be used 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, e.g., ?myField
- Virtual fields are specified with the suffix, e.g., ?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, where the last element is the pseudo-IRI lang(). The property preceding lang() must lead to a literal value. For example,

```graphdb
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": "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

The named graph of a given value can be indexed 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`.

**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 (e.g., http://www.ontotext.com/example/wine#Franvino from our small example dataset) indexed as a regular field. This can be achieved by specifying a property chain with a single property referring to the pseudo-IRI $self. For example,

```sql
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/wine#Wine"
    ],
    "fields": [
    {
      "fieldName": "entityId",
      "propertyChain": ["$self"
       ],
    },
    {
      "fieldName": "grape",
      "propertyChain": [
        "http://www.ontotext.com/example/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`.

5.14.3.6 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:
<table>
<thead>
<tr>
<th>RDF value</th>
<th>RDF datatype</th>
<th>Solr type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRI</td>
<td>n/a</td>
<td>string</td>
</tr>
<tr>
<td>literal</td>
<td>any type not explicitly mentioned below</td>
<td>text_general</td>
</tr>
<tr>
<td>literal</td>
<td>with one of the language tags en, de, es, ru</td>
<td>text_xx where xx is language dependent</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:double</td>
<td>pdouble (single value), pdouble (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:float</td>
<td>pfloat (single value), pfloats (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:long</td>
<td>plong (single value), plongs (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:int</td>
<td>pint (single value), pints (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:dateTime</td>
<td>pdate (single value), pdates (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:date</td>
<td>pdate (single value), pdates (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYear</td>
<td>pdate (single value), pdates (multivalued)</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYearMonth</td>
<td>pdate (single value), pdates (multivalued)</td>
</tr>
</tbody>
</table>

The datatype mapping can be affected by the synchronization options, too. For example, a non-analysed 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-normalised data, e.g., the first value has no datatype but other values have.

It is therefore recommended to set *datatype* to a fixed value, e.g. xsd:date.

**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 format and are *proleptic* years, i.e., 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, we define the undetermined time zone as UTC, i.e., "2020-02-14T12:00:00+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. E.g., "2020+02:00"^^xsd:gYear (the year 2020 in the +02:00 timezone) will be normalized to
if strict timezone adherence is followed. We have chosen to ignore the timezone on any values that do not have an associated time value, e.g.:

- "2020-02-15+02:00"^^xsd:date
- "2020-02+02:00"^^xsd:gYearMonth
- "2020+02:00"^^xsd:gYear

All of the above will be treated as if they specified UTC as their timezone.

### 5.14.3.7 Entity filtering

The Solr connector supports three kinds of entity filters used to fine-tune the set of entities and/or 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 `FILTER()` inside a SPARQL query but not exactly the same. In them, each configured field can be referred to by prefixing it with a `?`, much like referring to a variable in SPARQL.

#### Types of filters

**Top-level value filter** The top-level value filter is specified via `valueFilter`. 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 and 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 via `documentFilter`. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. It can be used to enforce global document restrictions, e.g., 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 via `valueFilter` 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 `$this`. Other field names can be used to filter the current field’s value based on the value of another field, e.g., `$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*.

#### Filter operators

The filter operators are used to test if the value of a given field satisfies some condition:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?var in (value1, value2, ...)</code></td>
<td>Tests if the field <code>var</code>’s value is one of the specified values. Values are compared strictly unlike the similar SPARQL operator, i.e. for literals to match their datatype must be exactly the same (similar to how SPARQL <code>sameTerm</code> works). Values that do not match, are treated as if they were not present in the repository.</td>
</tr>
</tbody>
</table>

Example:

| `?status in ("active", "new")` |

Continued on next page
<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?var not in (value1, value2, ...)</td>
<td>The negated version of the in-operator.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>?status not in (&quot;archived&quot;)</td>
</tr>
<tr>
<td>bound(?var)</td>
<td>Tests if the field var has a valid value. This can be used to make the field compulsory.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>bound(?name)</td>
</tr>
<tr>
<td>isExplicit(?var)</td>
<td>Tests if the field var’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.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>isExplicit(?name)</td>
</tr>
<tr>
<td>?var = value (equal to)</td>
<td>RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The field var’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.</td>
</tr>
<tr>
<td>?var != value (not equal to)</td>
<td>Examples:</td>
</tr>
<tr>
<td>?var &gt; value (greater than)</td>
<td>Given that height’s value is &quot;150&quot;^^xsd:int and dateOfBirth’s value is &quot;1989-12-31&quot;^^xsd:date, then:</td>
</tr>
<tr>
<td>?var &gt;= value (greater than or equal to)</td>
<td>?height = &quot;150&quot;^^xsd:int is true</td>
</tr>
<tr>
<td>?var &lt; value (less than)</td>
<td>?height = &quot;150&quot;^^xsd:long is true</td>
</tr>
<tr>
<td>?var &lt;= value (less than or equal to)</td>
<td>?height = &quot;150&quot; is false</td>
</tr>
<tr>
<td></td>
<td>?height != &quot;151&quot;^^xsd:int is true</td>
</tr>
<tr>
<td></td>
<td>?height != &quot;150&quot; is true</td>
</tr>
<tr>
<td></td>
<td>?height &gt; &quot;150&quot;^^xsd:int is false</td>
</tr>
<tr>
<td></td>
<td>?height &gt;= &quot;150&quot;^^xsd:int is true</td>
</tr>
<tr>
<td></td>
<td>?dateOfBirth &lt; &quot;1990-01-01&quot;^^xsd:date is true</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex(?var, &quot;pattern&quot;)</td>
<td>Tests if the field var’s value matches the given regular expression pattern. If the “i” flag option is present, this indicates that the match operates in case-insensitive mode. Values that do not match are treated as if they were not present in the repository. Example: regex(?name, &quot;^mrs?&quot;, &quot;i&quot;)</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regex(?var, &quot;pattern&quot;, &quot;i&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr1</td>
<td></td>
<td>expr2</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr1 or expr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr1 &amp;&amp; expr2</td>
<td>Logical conjunction of expressions expr1 and expr2.</td>
<td>Examples: bound(?status) &amp;&amp; ?status in (&quot;active&quot;, &quot;new&quot;) bound(?status) and ?status in (&quot;active&quot;, &quot;new&quot;)</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr1 and expr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!expr</td>
<td>Logical negation of expression expr.</td>
<td>Examples: !bound(?company)</td>
</tr>
<tr>
<td>( expr )</td>
<td>Grouping of expressions</td>
<td>Example: (bound(?name) or bound(?company)) &amp;&amp; bound(?address)</td>
</tr>
</tbody>
</table>

### Filter modifiers

In addition to the operators, there are some constructions that can be used to 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)` is used for going to a previous level in a property chain. It can be applied recursively as many times as needed, e.g., `parent(parent(parent(?var)))` goes back in the chain three times. The effective value of `parent(?var)` can be used with the `in` or `not in` operator like this: `parent(?var) in (<urn:a>, <urn:b>)`, or in the `bound` operator like this: `parent(bound(?var))`.

**Accessing an element beyond the chain** The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) is used to access additional values that are accessible through the property `uri`. In essence, this construction corresponds to the triple pattern `value uri ?effectiveValue`, where `value` is a value bound by the field `var`. The effective value of `?var -> uri` can be used with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. It can be combined with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. The same construction can be applied to the `bound` operator like this: `bound(?company -> <urn:hasBranch>)`, or even combined with `parent()` like this: `bound(parent(?company) -> <urn:hasGroup>)`.

The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (alternatively, just `type`), which
Filtering by RDF graph  The construction graph(?var) is used for accessing 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(?x) is the recommended way.

The construction can be combined with parent() like this: graph(parent(?a)) in (<urn:a>).

Filtering by language tags  The construction lang(?var) is used for accessing 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"). The construction can be combined with parent() and an element beyond the chain like this: lang(parent(?a) -> <http://www.w3.org/2000/01/rdf-schema#label>) in ("en", "bg"). Literal values without language tags can be filtered by using an empty tag: "".

Current context variable $this  The special field variable $this (and not ?this, $?this, $?this) is used to refer to the current context. In the top-level value filter and the top-level document filter it refers to the document ID. In the per-field value filter it refers to the currently filtered field value.

ALL() quantifier  In the context of document-level filtering, a match is true if at least one of potentially many field values match, e.g., ?location = <urn:Europe> would return true if the document contains { “location”: ["<urn:Asia>", "<urn:Europe>"] }.

In addition to this, you can also use the ALL() quantifier when you need all values to match, e.g., ALL(?location) = <urn:Europe> would not match with the above document because <urn:Asia> does not match.

Entity filters and default values  Entity filters can be combined with default values in order to get more flexible behavior.

A typical use-case for an entity filter is having soft deletes, i.e., instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.

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 comparing the field value to 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 and the top-level 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 as 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 being indexed first.
Basic entity filter example

Given the following RDF data:

```rdf
@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"
exmple: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)
exmple: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
exmple:gamma
   rdf:type example:gadget ;
   example:name "Mary Syncless" ;
   example:city "Liverpool" .
```

If you create a connector instance such as:

```sql
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 entity :beta is not synchronized as it has no value for city.

To handle such cases, you can modify the connector configuration to specify a default value for city:

```json
...
  {
    "fieldName": "city",
    "propertyChain": ["http://www.ontotext.com/example#city"],
    "defaultValue": "London"
  }
...```

5.14. GraphDB Connectors
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.

**Advanced entity filter example**

Sometimes, data represented in RDF is not well suited to map directly to non-RDF. For example, if you have news articles and they can be tagged with different concepts (locations, persons, events, etc.), 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
  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 .
```

Now, if you map this data to Solr so that the property :taggedWith is mapped to separate fields taggedWithPerson
and taggedWithLocation according to the type of x (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": "$this -> type in (<http://www.ontotext.com/example2#Person>)" 
      },
      { "fieldName": "taggedWithLocation",
        "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
        "valueFilter": "$this -> type in (<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:

<table>
<thead>
<tr>
<th>Article IRI</th>
<th>Value in taggedWith-Person</th>
<th>Value in taggedWithLocation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Article1</td>
<td>:Einstein</td>
<td>:Berlin</td>
<td>: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 as it does not match the filter.</td>
</tr>
<tr>
<td>:Article2</td>
<td></td>
<td>:Berlin</td>
<td>:taggedWith has the value :Berlin. After the filter is applied, only taggedWithLocation is populated.</td>
</tr>
<tr>
<td>:Article3</td>
<td>:Mozart</td>
<td></td>
<td>:taggedWith has the value :Mozart. After the filter is applied, only taggedWithPerson is populated.</td>
</tr>
<tr>
<td>:Article4</td>
<td>:Mozart</td>
<td>:Berlin</td>
<td>:taggedWith has the values :Berlin and :Mozart. The filter leaves only the correct values in the respective fields.</td>
</tr>
<tr>
<td>:Article5</td>
<td></td>
<td></td>
<td>:taggedWith has no values. The filter is not relevant.</td>
</tr>
<tr>
<td>:Article6</td>
<td></td>
<td></td>
<td>:taggedWith has the value :Cannes-FF. The filter removes it as it does not match.</td>
</tr>
</tbody>
</table>
This can be checked 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 _:f .
  _:f 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`:

<table>
<thead>
<tr>
<th>?facetName</th>
<th>?facetValue</th>
<th>?facetCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>taggedWithLocation</td>
<td><a href="http://www.ontotext.com/example2#Berlin">http://www.ontotext.com/example2#Berlin</a></td>
<td>3</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Mozart">http://www.ontotext.com/example2#Mozart</a></td>
<td>2</td>
</tr>
<tr>
<td>taggedWithPerson</td>
<td><a href="http://www.ontotext.com/example2#Einstein">http://www.ontotext.com/example2#Einstein</a></td>
<td>1</td>
</tr>
</tbody>
</table>

### 5.14.3.8 Overview of connector predicates

The following diagram shows a summary of all predicates that can administrate (create, drop, check status) 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.
5.14.3.9 SolrCloud support

From GraphDB 8.0/Connectors 6.0, the Solr connector has SolrCloud support. SolrCloud is the distributed version of Solr, which offers index sharding, better scaling, fault tolerance, etc. 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.

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 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 have to be mentioned 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.

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 want to create a new collection, you have to 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 have to specify the configset name in the copyConfigsFrom parameter. If you do not specify it, 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 config sets, use the parameter with the name of the custom config set, i.e., **customConfigset**.

**Example:** Create SolrCloud connector query using a custom configset

```xml
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.ontotext.com/example/wine#Wine"
    "fields": [
        "fieldName": "grape",
        "propertyChain": ["http://www.ontotext.com/example/wine#madeFromGrape",
            "http://www.w3.org/2000/01/rdf-schema#label"
        ],
    },
    "fieldName": "sugar",
    "propertyChain": ["http://www.ontotext.com/example/wine#hasSugar"
    ],
    "multivalued": false
    },
    "fieldName": "year",
    "propertyChain": ["http://www.ontotext.com/example/wine#hasYear"
    ]
    }
    "...."
}
```

5.14.3.10 Caveats

**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 need to come before any predicates that fetch results.

The diagram in **Overview of connector predicates** provides a quick overview of the predicates.
5.14.3.11 Upgrading from previous versions

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.

5.14.4 Kafka GraphDB connector

Note: This feature requires a GraphDB Enterprise license.

5.14.4.1 Overview and features

The Kafka connector provides a means to synchronize changes to the RDF model to any Kafka consumer, 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 defined as 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:

- maintaining a Kafka topic that is always in sync with the data stored in GraphDB;
- multiple independent instances per repository;
- the entities for synchronization are 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's of the entities for synchronization;
– a list of languages for synchronization (the default is all languages);
– additional filtering by property and value.

Unlike the Elasticsearch, Solr, and Lucene connectors, the Kafka connector does not have a query interface since Kafka is a simple message queue and does not provide search functionality.

Each feature is described in detail below.

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.

5.14.4.2 Usage

All interactions with the Kafka GraphDB Connector are done through SPARQL queries.

There are three types of SPARQL queries:

• INSERT for creating and deleting connector instances;
• SELECT for listing connector instances and querying their configuration parameters;
• INSERT for storing data as part of the normal GraphDB data workflow.

In general, this corresponds to INSERT adds or modifies data and SELECT queries existing data.

Each connector implementation defines its own IRI prefix 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 this prefix, e.g., 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 prefix to avoid clashing with any of the command predicates. For Kafka, the instance prefix 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.

```sparql
@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.ontotext.com/example/wine#> .

wine:RoseWine rdfs:subClassOf wine:Wine .

wine:Merlo
    rdf:type wine:Grape ;
    rdfs:label "Merlo" .

wine: Cabernet Sauvignon
    rdf:type wine:Grape ;
    rdfs:label "Cabernet Sauvignon" .

wine: Cabernet Franc
```

(continues on next page)
5.14.4.3 Setup and maintenance

Prerequisites

Third-party component versions  This version of the Kafka GraphDB Connector uses Kafka version 2.8.0.
Creating a connector instance

Creating a connector instance is done by sending a SPARQL query with the following configuration data:

- the name of the connector instance (e.g., my_index);
- a Kafka node and topic to synchronize to;
- classes to synchronize;
- properties to synchronize.

The configuration data has to be provided as a JSON string representation and passed together with the create command.

If you create the connector via the Workbench, no matter which way you use, you will be presented with a pop-up screen showing you the connector creation progress.

Using the 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.
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 createConnector predicate, e.g., it creates a connector instance called my_index, which synchronizes the wines from the sample data above:

```
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.ontotext.com/example/wine#Wine"
    } ],
    "fields": [
      { "fieldName": "grape",
        "propertyChain": [ "http://www.ontotext.com/example/wine#madeFromGrape"
      ] },
      { "fieldName": "sugar",
        "propertyChain": [ "http://www.ontotext.com/example/wine#hasSugar"
      ] },
      { "fieldName": "year",
        "propertyChain": [ "http://www.ontotext.com/example/wine#hasYear"
      ] }
  }
}
```

(continues on next page)
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 and, in the example, it is only entities of the type http://www.ontotext.com/example/wine#Wine (and its subtypes). The "fields" key defines the mapping from RDF to Kafka. The basic building block is the property chain, i.e., 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 later used in the queries.

Grape is an example of a property chain composed of more than one property. First, we take the wine’s made-FromGrape property, the object of which is an instance of the type Grape, and then we take the rdfs:label of this instance. 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 is recommended 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 these 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 Kafka. It should be possible to configure most of them by supplying the relevant Kafka producer properties via the kafkaProducerConfig parameter.

**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 dropConnector predicate where the name of the connector instance has to be in the subject position, e.g., this removes the connector my_index:

```sparql
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 in case a normal delete 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, and check Force delete in the dialog box.
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 .
}
```

Listing available connector instances

In the Connectors management view

Existing Connector instances show under Existing connectors (below the New Connector button). Click the name of an instance to view its configuration and SPARQL query, or click the repair / delete icons to perform these operations.
With a SPARQL query

Listing connector instances returns all previously created instances. It is a SELECT query with the listConnectors predicate:

```sparql
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>

SELECT ?cntUri ?cntStr 
  { ?cntUri kafka:listConnectors ?cntStr . }
```

?cntUri is bound to the prefixed IRI of the connector instance that was used during creation, e.g., http://www.ontotext.com/connectors/kafka/instance#my_index, while ?cntStr is bound to a string, representing the part after the prefix, e.g., "my_index".
Instance status check

The internal state of each connector instance can be queried using a SELECT query and the connectorStatus predicate:

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>

SELECT ?cntUri ?cntStatus {
}
```

?cntUri is bound to the prefixed 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.

5.14.4.4 Working with data

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, i.e., you must simply execute standard SPARQL INSERT/DELETE queries. This is achieved by intercepting all changes in the plugin and determining which abstract documents need to be updated.

5.14.4.5 List of creation parameters

The creation parameters define how a connector instance is created by the :createConnector predicate. 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 index for tracking updates to chains)

The only way to index changes in data after the connector has been created is to repair (or drop/recreate) the connector.

**importGraph (boolean), optional, a virtual graph containing data from which to create the connector** 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: <http://www.ontotext.com/connectors/kafka#> control prefix is used. The data have to 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 GDB Workbench, this can be done by pasting them one after another in the SPARQL editor and putting a semicolon at the end of the first INSERT. This functionality requires read-only mode.

```
PREFIX kafka: <http://www.ontotext.com/connectors/kafka#>

INSERT {
  GRAPH kafka:graph {
    ... 
  }
} WHERE {
  ... 
};
```

(continues on next page)
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 functionality requires readonly mode.

detectFields (boolean), optional, detects fields

This mode introduces automatic field detection when creating a connector. Instead of providing fields: [ ... ] in the JSON, you can skip this and get automatic fields where each field will have a single predicate chain and its field name will be the same as the predicate.

In this mode, specifying types is optional too. 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 the 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 format of the node value is of the form http://hostname.domain:port, https:// is allowed too. No default value. Can be updated at runtime without having 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 without having to rebuild the index.

kafkaPlainAuthUsername (string), optional, supplies the username for Kafka SASL PLAIN authentication

No default value. Can be updated at runtime without having to rebuild the index.

kafkaPlainAuthPassword (string), optional, supplies the password for Kafka SASL PLAIN authentication

No default value. Can be updated at runtime without having to rebuild the index.

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 without having to rebuild the index.

bulkUpdateRequestSize (integer), controls the maximum request size (and consequently the maximum size per document) in

Default value is 1,048,576 (1 megabyte). Can be updated at runtime without having to rebuild the index.

authenticationConfiguratorClass optional, provides custom authentication behavior

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 without having 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 config keys, e.g., transactional.id, are not allowed here. No default. Can be updated at runtime without having to rebuild the index.
`kafkaIgnoreDeleteAll` (boolean), optional, a flag that, when selected, will not notify Kafka when all repository statements are removed. 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 propagating the Kafka config False by default. See also `Producer sharing` and `Conflict resolution`. Can be updated at runtime without having 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. This can be done 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. In addition, an empty range can be used 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 The field name transform defines an optional transformation of the field name.

    – `none`: The field name is supplied via the `fieldName` option.

    – `predicate`: The field name is derived from the full IRI of the last predicate of the chain, e.g., if the last predicate was `http://www.w3.org/2000/01/rdf-schema#label`, then the field name will be `http://www.w3.org/2000/01/rdf-schema#label` too.

    – `predicate.localName`: The field name is derived from the local name of the IRI of the last predicate of the chain, e.g., 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 defined as 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  The default value (defaultValue) provides means for specifying 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 that some conversions are always valid, for example a literal or an IRI to a string 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.

The mapping can be overridden through the property "datatype", 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, provide 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.
Updating parameters at runtime

As mentioned above, the following connector parameters can be updated at runtime without having to rebuild the index:

- `kafkaNode`
- `kafkaSSL`
- `kafkaProducerConfig`
- `kafkaCompressionType`
- `kafkaPlainAuthUsername`
- `kafkaPlainAuthPassword`
- `bulkUpdateBatchSize`
- `bulkUpdateRequestSize`
- `kafkaPropagateConfig`

This can be done by executing the following SPARQL update, here with examples for changing the user and password:

```
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"
  }
}
```

Special field definitions

Nested objects

Nested objects are JSON objects that are used as values in the main document or other nested objects (up to five levels of nesting is possible). They are defined with the `objectFields` option.

Having the following data consisting 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:John>

<urn:Mary>
  a <urn:Person> ;
  <urn:name> "Mary" ;
  <urn:gender> <urn:Female> ;
  <urn:age> 58 ;
  <urn:hasSpouse> <urn:John> ;
  <urn:hasChild> <urn:Billy> .
</urn:Mary>
```
We can create a nested objects index that consists of children and grandchildren with their corresponding fields defining their gender and age:

```json
{
    "fields": [
        {
            "fieldName": "name",
            "propertyChain": [
                "urn:name"
            ],
            "datatype": "xsd:string"
        },
        {
            "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": [
        "$self"
      ]
    },
    {
      "fieldName": "name",
      "propertyChain": [
        "urn:name"
      ]
    },
    {
      "fieldName": "age",
      "propertyChain": [
        "urn:age"
      ],
      "datatype": "xsd:long"
    },
    {
      "fieldName": "gender",
      "propertyChain": [
        "urn:gender",
        "urn:label"
      ]
    },
    {
      "fieldName": "children",
      "propertyChain": [
        "urn:hasChild"
      ],
      "objectFields": [
        {
          "fieldName": "id",
          "propertyChain": [
            "$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"
    ]
  },
  {
    "fieldName": "age",
    "propertyChain": [
      "urn:age"
    ],
    "datatype": "xsd:long"
  },
  {
    "fieldName": "gender",
    "propertyChain": [
      "urn:gender",
      "urn:label"
    ]
  }
]

"types": [
  "urn:Person"
],
"kafkaNode": ....,
"kafkaTopic": ...
Copy fields

Often, it is 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. This is achieved by specifying 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",  
    "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, e.g., 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 alphanumeric sequence of convenience. 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"  
    ]  
  }  
],  
...  
...  
```
The values of the fields name$1 and name$2 will be merged and synchronized to the field name in Kafka.

**Note:** You cannot mix suffixed and unsuffixed fields with the same name, e.g., if you defined myField$new and myField$old, you cannot have a field called just myField.

**Filters and fields with multiple property chains**

Filters can be used 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, e.g., ?myField
- Virtual fields are specified with the suffix, e.g., ?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, where the last element is the pseudo-IRI lang(). The property preceding lang() must lead to a literal value. For example:

```sql
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.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

The named graph of a given value can be indexed 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.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`.

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 (e.g., http://www.ontotext.com/example/wine#Franvino from our small example dataset) indexed as a regular field. This can be achieved 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.ontotext.com/example/wine#Wine"
  ],
  "fields": [
    "fieldName": "entityId",
    "propertyChain": [ "$self"
    ],
    "fieldName": "grape",
    "propertyChain": [ "http://www.ontotext.com/example/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`.

5.14.4.6 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 vs. RDF datatype vs. JSON type

<table>
<thead>
<tr>
<th>RDF value</th>
<th>RDF datatype</th>
<th>JSON type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRI</td>
<td>n/a</td>
<td>string</td>
</tr>
<tr>
<td>literal</td>
<td>any type not explicitly mentioned below</td>
<td>string</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:double</td>
<td>number</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:float</td>
<td>number</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:long</td>
<td>number</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:int</td>
<td>number</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:dateTime</td>
<td>string in ISO format with time zone</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:date</td>
<td>string in ISO format without time zone</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:time</td>
<td>string in ISO format with time zone</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYear</td>
<td>string in ISO format without time zone</td>
</tr>
<tr>
<td>literal</td>
<td>xsd:gYearMonth</td>
<td>string in ISO format without time zone</td>
</tr>
</tbody>
</table>

**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, e.g., the first value has no datatype but other values have.

It is therefore recommended to set **datatype** to a fixed value, e.g., `xsd:date`.

### 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, i.e., 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, we define the undetermined time zone as UTC, i.e., "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. E.g., "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 strict timezone adherence is followed. We have chosen to ignore the timezone on any values that do not have an associated time value, e.g.:

- "2020-02-15T02:00:00"^^xsd:date
- "2020-02+02:00"^^xsd:gYearMonth
- "2020+02:00"^^xsd:gYear
All of the above will be treated as if they specified UTC as their timezone.

5.14.4.7 Entity filtering

The Kafka connector supports four kinds of entity filters used to fine-tune the set of entities and/or 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 `FILTER()` inside a SPARQL query but not exactly the same. In them, each configured field can be referred to by prefixing it with a `?`, much like referring to a variable in SPARQL.

Types of filters

**Top-level value filter** The top-level value filter is specified via `valueFilter`. 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 and 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 via `documentFilter`. This filter is evaluated last when all of the document has been collected and it decides whether to include the document in the index. It can be used to enforce global document restrictions, e.g., 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 via `valueFilter` 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 `$this`. Other field names can be used to filter the current field’s value based on the value of another field, e.g., `$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 via `documentFilter` 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 we have a field `children` that defines a nested document, and we use a filter like `?age < "10"^^xsd:int`, we will be referring to the field `children.age`. We can use the prefix `Souter` one or more times to refer to field values from the outer document (from the viewpoint of the nested document). For example, `Souter.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.
Filter operators

The filter operators are used to test if the value of a given field satisfies some condition:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?var in (value1, value2, ...)</code></td>
<td>Tests if the field <code>var</code>’s value is one of the specified values. Values are compared strictly unlike the similar SPARQL operator, i.e. for literals to match their datatype must be exactly the same (similar to how SPARQL <code>sameTerm</code> works). Values that do not match, are treated as if they were not present in the repository. Example: <code>?status in (&quot;active&quot;, &quot;new&quot;)</code></td>
</tr>
<tr>
<td><code>?var not in (value1, value2, ...)</code></td>
<td>The negated version of the <code>in</code>-operator. Example: <code>?status not in (&quot;archived&quot;)</code></td>
</tr>
<tr>
<td><code>bound(?var)</code></td>
<td>Tests if the field <code>var</code> has a valid value. This can be used to make the field compulsory. Example: <code>bound(?name)</code></td>
</tr>
<tr>
<td><code>isExplicit(?var)</code></td>
<td>Tests if the field <code>var</code>’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 <code>parent(?var)</code> as many times as needed. Example: <code>isExplicit(?name)</code></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 6 – continued from previous page

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?var = value (equal to)</td>
<td>RDF value comparison operators that compare RDF values similarly to the equivalent SPARQL operators. The field var’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’s value is &quot;150&quot;^^xsd:int and dateOfBirth’s value is &quot;1989-12-31&quot;^^xsd:date, then:</td>
</tr>
<tr>
<td>?var != value (not equal to)</td>
<td>?height = &quot;150&quot;^^xsd:int is true</td>
</tr>
<tr>
<td>?var &gt; value (greater than)</td>
<td>?height = &quot;150&quot;^^xsd:long is true</td>
</tr>
<tr>
<td>?var &gt;= value (greater than or equal to)</td>
<td>?height = &quot;150&quot; is false</td>
</tr>
<tr>
<td>?var &lt; value (less than)</td>
<td>?height != &quot;151&quot;^^xsd:int is true</td>
</tr>
<tr>
<td>?var &lt;= value (less than or equal to)</td>
<td>?height != &quot;150&quot; is true</td>
</tr>
<tr>
<td>regex(?var, &quot;pattern&quot;)</td>
<td>Tests if the field var’s value matches the given regular expression pattern. If the “i” flag option is present, this indicates that the match operates in case-insensitive mode. Values that do not match are treated as if they were not present in the repository. Example:</td>
</tr>
<tr>
<td>or</td>
<td>regex(?name, &quot;^mrs?$&quot;, &quot;i&quot;)</td>
</tr>
<tr>
<td>regex(?var, &quot;pattern&quot;, &quot;i&quot;)</td>
<td></td>
</tr>
<tr>
<td>expr1</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>expr1 or expr2</td>
</tr>
<tr>
<td>expr1 &amp;&amp; expr2</td>
<td>Logical conjunction of expressions expr1 and expr2. Examples: bound(?status) &amp;&amp; ?status in (&quot;active&quot;, &quot;new&quot;) or bound(?status) and ?status in (&quot;active&quot;, &quot;new&quot;)</td>
</tr>
<tr>
<td>or</td>
<td>expr1 and expr2</td>
</tr>
<tr>
<td>!expr</td>
<td>Logical negation of expression expr. Example: !bound(?company)</td>
</tr>
</tbody>
</table>

Continued on next page
Filter modifiers

In addition to the operators, there are some constructions that can be used to 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)` is used for going to a previous level in a property chain. It can be applied recursively as many times as needed, e.g., `parent(parent(parent(?var)))` goes back in the chain three times. The effective value of `parent(?var)` can be used with the `in` or `not in` operator like this: `parent(?company) in (<urn:a>, <urn:b>)`, or in the `bound` operator like this: `parent(bound(?var))`.

**Accessing an element beyond the chain** The construction `?var -> uri` (alternatively, `?var o uri` or just `?var uri`) is used to access additional values that are accessible through the property `uri`. In essence, this construction corresponds to the triple pattern `value uri ?effectiveValue`, where `?value` is a value bound by the field `var`. The effective value of `?var -> uri` can be used with the `in` or `not in` operator like this: `?company -> rdf:type in (<urn:c>, <urn:d>)`. It can be combined with `parent()` like this: `parent(?company) -> rdf:type in (<urn:c>, <urn:d>)`. The same construction can be applied to the `bound` operator like this: `bound(?company) -> <urn:hasBranch>`, or even combined with `parent()` like this: `bound(parent(?company)) -> <urn:hasGroup>`. The IRI parameter can be a full IRI within `< >` or the special string `rdf:type` (alternatively, 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)` is used for accessing 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(?x)` is the recommended way.

The construction can be combined with `parent()` like this: `graph(parent(?a)) in (<urn:a>)`.

**Filtering by language tags** The construction `lang(?var)` is used for accessing 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")`. The construction can be combined with `parent()` and an element beyond the chain like this: `lang(parent(?a)) -> <http://www.w3.org/2000/01/rdf-schema#label> in ("en", "bg")`. Literal values without language tags can be filtered by using an empty tag: "".

**Current context variable $this** The special field variable `$this` (and not `?this`, `?$this`, `??this`) is used to refer to the current context. In the top-level value filter and the top-level document filter it refers to the document ID. 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 ID.

**ALL() quantifier** In the context of document-level filtering, a match is `true` if at least one of potentially many field values match, e.g., `?location = <urn:Europe>` would return `true` if the document contains `{ "location" : ["<urn:Asia>", "<urn:Europe>"] }`.

In addition to this, you can also use the `ALL()` quantifier when you need all values to match, e.g., `ALL( ?location) = <urn:Europe>` would not match with the above document because `<urn:Asia>` does not match.

**Entity filters and default values** Entity filters can be combined with default values in order to get more flexible behavior.

A typical use-case for an entity filter is having soft deletes, i.e., instead of deleting an entity, it is marked as deleted by the presence of a specific value for a given property.
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 comparing the field value to 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 as 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 being indexed first.

Basic entity filter example

Given the following RDF data:

```_reasonable
@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")
ex\ample:alpha
    rdf:ty\pe \example:gadget ;
    \exam\ple:name "John Synced" ;
    \example:city "London" .
# the entity below will not be synchronised because it lacks the property completely: bound(?city)
ex\ample:beta
    rdf:ty\pe \example:gadget ;
    \exam\ple:name "Peter Syncfree" .
# the entity below will not be synchronised because it has a different city value:
# ?city in ("London") will remove the value "Liverpool" so bound(?city) will be false
ex\ample:gamma
    rdf:ty\pe \example:gadget ;
    \exam\ple:name "Mary Syncless" ;
    \example:city "Liverpool" .
```

If you create a connector instance such as:

```reasonable
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.ontotext.com/example#gadget"],
    "fields": |
```

(continues on next page)
The entity :beta is not synchronized as it has no value for city.

To handle such cases, you can modify the connector configuration to specify a default value for city:

```json

{  "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.

Advanced entity filter example

Sometimes, data represented in RDF is not well suited to map directly to non-RDF. For example, if you have news articles and they can be tagged with different concepts (locations, persons, events, etc.), one possible way to model this is a single property :taggedWith. Consider the following RDF data:

```r
@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  
    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)
Now, if you map this data to Kafka so that the property :taggedWith is mapped to separate fields taggedWithPerson and taggedWithLocation according to the type of x (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:

```sql
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:9992",
    "kafkaTopic": "my_index",
    "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": "$this -> type in ("http://www.ontotext.com/example2#Person")"
      },
      {
        "fieldName": "taggedWithLocation",
        "propertyChain": ["http://www.ontotext.com/example2#taggedWith"],
        "valueFilter": "$this -> type in ("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:

<table>
<thead>
<tr>
<th>Article IRI</th>
<th>Value in taggedWith-Person</th>
<th>Value in taggedWith-Locaton</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Article1</td>
<td>:Einstein</td>
<td>:Berlin</td>
<td>: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 as it does not match the filter.</td>
</tr>
<tr>
<td>:Article2</td>
<td></td>
<td>:Berlin</td>
<td>:taggedWith has the value :Berlin. After the filter is applied, only taggedWithLocation is populated.</td>
</tr>
<tr>
<td>:Article3</td>
<td>:Mozart</td>
<td></td>
<td>:taggedWith has the value :Mozart. After the filter is applied, only taggedWithPerson is populated</td>
</tr>
<tr>
<td>:Article4</td>
<td>:Mozart</td>
<td>:Berlin</td>
<td>:taggedWith has the values :Berlin and :Mozart. The filter leaves only the correct values in the respective fields.</td>
</tr>
<tr>
<td>:Article5</td>
<td></td>
<td></td>
<td>:taggedWith has no values. The filter is not relevant.</td>
</tr>
<tr>
<td>:Article6</td>
<td></td>
<td></td>
<td>:taggedWith has the value :Cannes-FF. The filter removes it as it does not match.</td>
</tr>
</tbody>
</table>

5.14.4.8 Overview of connector predicates

The following diagram shows a summary of all predicates that can administrate (create, drop, check status) 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.
5.14.4.9 Caveats

Producer sharing

The Kafka connector aims to minimize resource usage and provide smooth transactional operation. This is achieved 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:

- Memory consumption is reduced as 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, e.g., they must have the same values for the `bulkUpdateBatchSize` and `kafkaCompressionType` options.

Once you have created at least one Kafka connector instance and attempt 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

When 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.

5.14.4.10 Upgrading from previous versions

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:

   ![Filter Example]

   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.

5.14.5 MongoDB integration

5.14.5.1 Overview and features

The MongoDB integration feature is a GraphDB plugin allowing users to query MongoDB databases using SPARQL and to execute heterogeneous joins. This section describes how to configure GraphDB and MongoDB to work together.

MongoDB is a document-based database with the biggest developer/user community. It is part of the MEAN technology stack and guarantees scalability and performance well beyond the throughput supported in GraphDB. Often, we see use cases with extreme scalability requirements and simple data model (i.e., tree representation of a document and its metadata).

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 RDF model.
Each feature is described in detail below.

5.14.5.2 Usage

The 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 valid JSON-LDs.

The JSON-LD documents are in hierarchical view allowing more complex search querying of embedded/nested documents.

Each document can be in 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 in GraphDB, in order to make inferred statements. The import of parts is an option for future development.

Below is shown 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_Briculaire_de_la_Dixmerie"
      },
      {
        "@id": "dbpedia:Bernard_Roberts"
      },
      {
        "@id": "dbpedia:Bartolomé_de_Medina"
      },
      {
        "@id": "dbpedia:Don_Bonker"
      },
      {
        "@id": "dbpedia:Cornel_Nistorescu"
      }
    ]
  }
}
```

(continues on next page)
{ "@id": "dbpedia:Clete_Roberts"
},
{ "@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 altText 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 disbelief 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#",
  "cwork": "http://www.bbc.co.uk/context/1#id"
}
• _id key is a MongoDB internal key.
• @graph node represents the RDF context in the JSON-LD doc.
• @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 to RDF model. The dates are extended for faster search/sorting. The ISODate in MongoDB is its internal way to store dates and is optimized for searching. This step will make querying/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.
5.14.5.3 Setup and maintenance

Installing MongoDB

Setting up and maintaining a MongoDB database is a separate task and must be accomplished outside of GraphDB. See the MongoDB website for details.

**Note:** Throughout the rest of this document, we assume that you have the MongoDB server installed and running on a computer you can access.

**Note:** 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 on the MongoDB website.

Creating an index

To configure GraphDB with MongoDB connection settings, we 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 is a sample query of how to create a MongoDB index:

```prefix
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.
Upgrading an index

When upgrading to a newer GraphDB version, it might happen that it contains plugins that are not present in the older version. In this case, 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 in question by executing the SPARQL query provided above, and also make sure to not delete the database in the plugin you are using.

Deleting an index

Deletion of an index is done using the following query:

```
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX mongodb-index: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
  mongodb-index:spb1000 :drop _:b .
}
```

Loading sample data

Import this `cwork1000.json` file with 1,000 of CreativeWork documents in MongoDB database “ldbc” and “creativeWorks” collection.

```
mongoimport --db ldbc --collection creativeWorks --file cwork1000.json
```

Querying MongoDB

This is a sample query that returns the `dateModified` for docs with the specific 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 {
  }
}
```
In a query, use the exact values as in the docs. 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 BSON 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:

- :find - accepts single BSON and sets a query string. The value is used to call `db.collection.find()`.
- :project - accepts single BSON. The value is used to select the projection for the results returned by :find. Find more info at MongoDB: Project Fields to Return from Query.
- :aggregate - accepts an array of BSONs. Calls `db.collection.aggregate()`. This is the most flexible way to make a MongoDB query as the `find()` method is just a single phase of the aggregation pipeline. The :aggregate predicate takes precedence over :find and :project. This means that if both :aggregate and :find are used, :find will be ignored.
- :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.
- :entity - (required) 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 @id node is used. Even if the value from this predicate is not used, it is required to have it in the query in order to inform the plugin that the graph part of the current iteration is completed.
- :hint - specifies the index to be used when executing the query (calls `cursor.hint()`).
- :collation - (optional) accepts BSON. Specifies language-specific rules for string comparison, such as rules for lettercase and accent marks. It is applied to a :find or an :aggregate query.
Multiple index calls in the same query

Multiple MongoDB calls are supported in the same query. 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:

```sparql
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:

```sparql
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 ;
    mongodb:find '{@graph.@id" : "http://www.bbc.co.uk/things/2#id"}';
    mongodb:entity ?entity .
    GRAPH mongodb:search2 {
      ?creativeWork cwork:dateModified ?modified ;
    }
  }
}
```

(continues on next page)
Both examples return the same result.

<table>
<thead>
<tr>
<th>creativeWork</th>
<th>modified</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.bbc.co.uk/things/1#id">http://www.bbc.co.uk/things/1#id</a></td>
<td>&quot;2012-02-14T12:43:13.165+02:00&quot;^^xsd:dateTime</td>
</tr>
<tr>
<td><a href="http://www.bbc.co.uk/things/2#id">http://www.bbc.co.uk/things/2#id</a></td>
<td>&quot;2011-11-27T13:17:29.243+02:00&quot;^^xsd:dateTime</td>
</tr>
</tbody>
</table>

Using aggregation functions

MongoDB has a number of aggregation functions such as: min, max, size, etc. These functions are called using the :aggregate predicate. The data of the retrieved results has to be converted to 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 really flexible but is prone to error.

Let’s examine the following query:

```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 .  
      }  
      ?s mongodb-index:graph ?g2 .  
    }  
  }```

5.14. GraphDB Connectors
It looks really similar to the first one except that instead of `@graph.cwork:graph.@id` we are writing the value to `@graph.inst:graph.@id` and as a result `?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.

Custom fields

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#>

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:index:entity ?entity .
  GRAPH mongodb-index:spb1000 {
    ?s mongodb-index:size ?size ;
    mongodb-index:halfSize ?halfSize .
  }
}
```

<table>
<thead>
<tr>
<th>size</th>
<th>halfSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>342</td>
<td>171.0</td>
</tr>
</tbody>
</table>

The values are projected as child elements of a custom node. After JSON-LD is taken from MongoDB, a preprocessing 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.

### Authentication

All types of authentication can be achieved by setting the credentials in the connection string. However, as it is not a good practice to store the passwords in plain text, the `:user`, `:password`, and `:authDb` predicates are introduced. If one of those predicates is used, it is mandatory to 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 as no passwords are being stored.

#### 5.14.6 General full-text search with the connectors

The GraphDB Connectors offer an excellent solution for indexing data with a well-known schema, e.g., index documents that have type A, where each document has a field F1 that can be reached by following the property chain composed of IRIs P1 and P2.

The features described below add a more general full-text search functionality to the connectors, and can be used individually or combined as desired to meet the specific needs of the use case.
5.14.6.1 Useful connector features

The following connector features are useful when defining a connector for general full-text search:

**Wildcard literal**

This feature allows for indexing of 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](#), and the [Elasticsearch connector](#).

**Field names derived from the predicate**

This feature allows for having 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](#), and the [Elasticsearch connector](#).

**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](#), and the [Elasticsearch connector](#).

5.14.6.2 Examples

All examples use the Star Wars RDF dataset. Download [starwars-data.ttl](#) and import it into a fresh repository before proceeding further.

The example connector definitions use the Lucene connector but can be easily adapted to Solr and Elasticsearch by changing `lucene` in the prefix definitions to `solr` or `elasticsearch`, and adding any additional parameters required by the respective connector, e.g., `elasticsearchNode`.

**Indexing all literals**

To index all literals in the repository regardless of where they are attached in the graph, you can combine **wildcard literal** and **untyped indexing**. Create a connector such as:

```json
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": [""
  }
}
```

(continues on next page)
The connector defines a single field, $fts$, that will index all literals regardless of their predicate: $\text{literal}$ 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: $\text{untyped}$ in the $\text{types}$ parameter.

Since the Star Wars dataset contains literals in many different languages, we restrict the index definition further by specifying "" (the empty language = any literal without a language tag) using the $\text{languages}$ option.

We can now search in this connector as usual, for example for the FTS query “luke skywalker”:

```sparql
# Full-text search for "skywalker"
SELECT ?entity ?label {
  { a con-inst:starwars_fts ;
    con:query "skywalker" ;
    con:entities ?entity .
  } FILTER(lang(?label) = "")
}
```

We get many different results belonging to different types (showing only the first ten results):

<table>
<thead>
<tr>
<th>?entity</th>
<th>?label</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://swapi.co/resource/human/43">https://swapi.co/resource/human/43</a></td>
<td>Shmi Skywalker</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/1">https://swapi.co/resource/human/1</a></td>
<td>Luke Skywalker</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/35">https://swapi.co/resource/human/35</a></td>
<td>Padmé Amidala</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/planet/1">https://swapi.co/resource/planet/1</a></td>
<td>Tatooine</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/18">https://swapi.co/resource/human/18</a></td>
<td>Obi-Wan Kenobi</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/11">https://swapi.co/resource/human/11</a></td>
<td>Anakin Skywalker</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/droid/2">https://swapi.co/resource/droid/2</a></td>
<td>C-3PO</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/4">https://swapi.co/resource/human/4</a></td>
<td>Darth Vader</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/droid/3">https://swapi.co/resource/droid/3</a></td>
<td>R2-D2</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/18">https://swapi.co/resource/human/18</a></td>
<td>Wedge Antilles</td>
</tr>
</tbody>
</table>

**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. It can be fine-tuned by using field names derived from the predicate. In this example, we added ""$\text{fieldNameTransform}$": "$\text{predicate.localName}$" 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 ""
}
```
We can use this connector to do general full-text searches, but also more precise ones, such as a query only in the label of entities (the field label is the result of taking the local name of <http://www.w3.org/2000/01/rdf-schema#label> at indexing time):

```sparql
# 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" ;
    con:entities ?entity .
  ?entity rdfs:label ?label
  FILTER(lang(?label) = "")
}
```

We get only three results back, namely the people that have “Skywalker” in their name:

<table>
<thead>
<tr>
<th>?entity</th>
<th>?label</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://swapi.co/resource/human/43">https://swapi.co/resource/human/43</a></td>
<td>Shmi Skywalker</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/1">https://swapi.co/resource/human/1</a></td>
<td>Luke Skywalker</td>
</tr>
<tr>
<td><a href="https://swapi.co/resource/human/11">https://swapi.co/resource/human/11</a></td>
<td>Anakin Skywalker</td>
</tr>
</tbody>
</table>

5.15 SPARQL Federation

5.15.1 Overview

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.

For example, to discover DBpedia resources about people who have the same names as those stored in a local repository, use the following query:
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 and, if so, returns the ID.

**Note:** Federation must be used with caution. First of all, to avoid doing 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 a similar rdfs:label to the rdfs:label of `<http://dbpedia.org/resource/Vaccination>` in the first SPARQL endpoint:

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?endpoint2_id {
    SERVICE <http://faraway_endpoint.org/sparql> {
        FILTER ( langMatches( lang(?l1), "en") )
    }
    SERVICE <http://remote_endpoint.com/sparql> {
        FILTER ( str(?l2) = str(?l1) )
    }
    BINDINGS ?endpoint1_id {
        { <http://dbpedia.org/resource/Vaccination> }
    }
}
```

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.

Query execution can be optimized by batching multiple values where the following is valid:
- 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.

### 5.15.2 Internal SPARQL federation

Since RDF4J repositories are also SPARQL endpoints, it is possible to use the federation mechanism to do distributed querying over several repositories on a local server. You can do it 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. The 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`.

### 5.15.2.1 Usage

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 repository you want to access. For example, to access the repository `authors` via internal federation, use a query like this:

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {
  books:author ?author .

  SERVICE <repository:authors> {
    ?author rdfs:label ?authorName
  }
}
```

The approach applied for DBpedia, `SERVICE <http://localhost:7200/repositories/my_labels>`, is also valid, but is less efficient.

### 5.15.2.2 Parameters

There are four parameters that control how the federated part of the query is executed:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>infer</code> (boolean)</td>
<td>Controls if inferred statements are included. True by default.</td>
</tr>
<tr>
<td></td>
<td>When set to <code>false</code>, it is equivalent to adding <code>FROM &lt;http://www.ontotext.com/explicit&gt;</code> to the federated query.</td>
</tr>
<tr>
<td><code>sameAs</code> (boolean)</td>
<td>Controls if statements are expanded over <code>owl:sameAs</code>. True by default.</td>
</tr>
<tr>
<td></td>
<td>When set to <code>false</code>, it is equivalent to adding <code>FROM &lt;http://www.ontotext.com/disable-sameAs&gt;</code> to the federated query.</td>
</tr>
<tr>
<td><code>from</code> (string)</td>
<td>Can be repeated multiple times, translates to <code>FROM &lt;...&gt;</code> No default value.</td>
</tr>
<tr>
<td><code>fromNamed</code> (string)</td>
<td>Can be repeated multiple times, translates to <code>FROM NAMED &lt;...&gt;</code> No default value.</td>
</tr>
</tbody>
</table>

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.
- `service <repository:repo1>` No `FROM` and `FROM NAMED`.

### 5.15. SPARQL Federation

537
service <repository:repo1, from=http://test.com, fromNamed=http://test.com/named, sameAs=false>


**Note:** This needs to be a valid URL and thus there cannot be spaces/blanks.

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:

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {
  books:author ?author .

  SERVICE <repository:authors, infer=false, sameAs=false> {
    ?author rdfs:label ?authorName
  }
}
```

### 5.15.3 Federated query to a remote password-protected repository

You can also use federation to query a remote password-protected GraphDB repository. There are two ways to do this:

- **By editing the repository configuration as follows:**
  1. Download the configuration file.
  2. In it, edit the `repositoryURL` (<http://user:password@localhost:7200/repositories/<RepositoryName>)) by placing your login details and the remote repository name.
  3. Stop GraphDB if it is running.
  4. Create a new directory in `$GDB_HOME/data/repositories/` with the same name as `repositoryID` from the config file.
  5. Place the edited config file in the newly created folder. Make sure that it is named `config.ttl`, as otherwise GraphDB will not recognize it and the repository will not be created.
  6. Start GraphDB again.

- **By importing the repository configuration file in the Workbench (does not require stopping GraphDB):**
  1. Download the mentioned configuration file.
  2. In it, change `rep:repositoryID"<RepoName>"` to the name of your repository.
  3. Edit the `repositoryURL` (<http://user:password@localhost:7200/repositories/<RepositoryName>)) by placing your login details and the remote repository name.
  4. Open GraphDB Workbench and go to Repositories > Create new repository > Create from file.
  5. Upload the file. The newly created repository will have the same name used for `<RepoName>`.

This will enable you to query the remote repository like in the above example:

```sparql
SELECT ?id ?label
WHERE {
  ?id a ex:Concept .
  SERVICE <repository:my_labels> {
    (continues on next page)
    }
```
Any URL parameters supported by the remote endpoint can be used, e.g., if it is an RDF4J/GraphDB repository, it could be a URL like `http://factforge.net/repositories/ff-news?infer=false` to include only explicit statements.

### 5.16 FedX Federation

#### 5.16.1 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.

In the context of 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 Sesame framework with a federation layer and is incorporated in it as Sail (Storage and Inference Layer).

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

#### 5.16.2 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 being in the same database.

- **Federation of sharded knowledge graphs:** A virtual knowledge graph can consist of various knowledge subgraphs distributed in a separate endpoint and can be virtually integrated using FedX Join. FedX is optimized for such scenarios where each graph has a different schema, i.e., 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**
5.16.3 Usage scenarios

In the following sections, we will demonstrate how semantic technology in the context of FedEx federation lowers the cost of searching and analyzing data sources by implementing a two-step integration process: (1) mapping any dataset to an ontology and (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.

5.16.3.1 Linked data cloud exploration

The first type of use case that we will look at is creating 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, i.e., predicates or types are frequently repeated across the different sources.

Note: Keep in mind that bnodes are not supported between FedEx members.

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

1. Create a FedEx repository via Setup Repositories Create new repository FedEx Virtual SPARQL.
2. In the configuration screen that you are taken to, click Add remote repository.
3. From the endpoint options in the dialog, 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.
7. Save the repository and connect it.

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”.

```sparql
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.
  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”.

These will be the first 15 house cats returned:

<table>
<thead>
<tr>
<th>#</th>
<th>URI</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td>&quot;CC&quot;</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.wikidata.org/entity/Q85549449">http://www.wikidata.org/entity/Q85549449</a></td>
<td>&quot;Unimark Barr&quot;</td>
</tr>
<tr>
<td>3</td>
<td><a href="http://www.wikidata.org/entity/Q11900883">http://www.wikidata.org/entity/Q11900883</a></td>
<td>&quot;Camomile&quot;</td>
</tr>
<tr>
<td>4</td>
<td><a href="http://www.wikidata.org/entity/Q1201902">http://www.wikidata.org/entity/Q1201902</a></td>
<td>&quot;Tame&quot;</td>
</tr>
<tr>
<td>5</td>
<td><a href="http://www.wikidata.org/entity/Q1637234">http://www.wikidata.org/entity/Q1637234</a></td>
<td>&quot;Humphrey&quot;</td>
</tr>
<tr>
<td>6</td>
<td><a href="http://www.wikidata.org/entity/Q1419629">http://www.wikidata.org/entity/Q1419629</a></td>
<td>&quot;Hare&quot;</td>
</tr>
<tr>
<td>7</td>
<td><a href="http://www.wikidata.org/entity/Q1629588">http://www.wikidata.org/entity/Q1629588</a></td>
<td>&quot;Hodge&quot;</td>
</tr>
<tr>
<td>8</td>
<td><a href="http://www.wikidata.org/entity/Q677525">http://www.wikidata.org/entity/Q677525</a></td>
<td>&quot;Orange&quot;</td>
</tr>
<tr>
<td>9</td>
<td><a href="http://www.wikidata.org/entity/Q651190">http://www.wikidata.org/entity/Q651190</a></td>
<td>&quot;Mrs. Chippy&quot;</td>
</tr>
<tr>
<td>10</td>
<td><a href="http://www.wikidata.org/entity/Q467667">http://www.wikidata.org/entity/Q467667</a></td>
<td>&quot;Muttz&quot;</td>
</tr>
<tr>
<td>11</td>
<td><a href="http://www.wikidata.org/entity/Q1207136">http://www.wikidata.org/entity/Q1207136</a></td>
<td>&quot;Dewey Readmore books&quot;</td>
</tr>
<tr>
<td>12</td>
<td><a href="http://www.wikidata.org/entity/Q1971146">http://www.wikidata.org/entity/Q1971146</a></td>
<td>&quot;Bocks&quot;</td>
</tr>
<tr>
<td>13</td>
<td><a href="http://www.wikidata.org/entity/Q196221">http://www.wikidata.org/entity/Q196221</a></td>
<td>&quot;F. O. C. Weirdest&quot;</td>
</tr>
<tr>
<td>14</td>
<td><a href="http://www.wikidata.org/entity/Q1759097">http://www.wikidata.org/entity/Q1759097</a></td>
<td>&quot;Toward&quot;</td>
</tr>
<tr>
<td>15</td>
<td><a href="http://www.wikidata.org/entity/Q1799624">http://www.wikidata.org/entity/Q1799624</a></td>
<td>&quot;Weirdestforce&quot;</td>
</tr>
</tbody>
</table>

### 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.

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

### Scenario 3: Querying both endpoints for a specific resource

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

<table>
<thead>
<tr>
<th>subject</th>
<th>predicate</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/version">http://schema.org/version</a></td>
<td>&quot;131678074-64bb46f4c4b8&quot;</td>
</tr>
<tr>
<td>2 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/lastModified">http://schema.org/lastModified</a></td>
<td>&quot;2020-12-27T13:20:00Z&quot;</td>
</tr>
<tr>
<td>3 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kot&quot;</td>
</tr>
<tr>
<td>4 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>5 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>6 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>7 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>8 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>9 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>10 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>11 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>12 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>13 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>14 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://schema.org/description">http://schema.org/description</a></td>
<td>&quot;kitten&quot;</td>
</tr>
<tr>
<td>15 <a href="http://www.wikidata.org/entity/Q378619">http://www.wikidata.org/entity/Q378619</a></td>
<td><a href="http://www.w3.org/2000/01/rdf-schema#label">http://www.w3.org/2000/01/rdf-schema#label</a></td>
<td>&quot;CC&quot;</td>
</tr>
</tbody>
</table>
Let’s explore both DBpedia and Wikidata for products made by the company Amazon.

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

CONSTRUCT
{
    ?wCompany owl:sameAs ?wCompany .
} WHERE {
    BIND( dbr:Amazon\(company\) as ?dbpCompany)

    # Get all products from DBpedia
} UNION {
    # Get all products from Wikidata
    ?dbpCompany owl:sameAs ?wCompany .
}

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 will then be able to explore any resource that we input.

With the following steps, create a graph config 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. The default Starting point is Start with a search box. In it, select the Graph expansion tab and enter the following query:

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

CONSTRUCT{(continues on next page)
The two databases are connected through 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 clear it up a little by pulling the node labels. To do so, go to the Node basics tab and enter the query:

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT ?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. You will be taken back to the Visual graph screen where the newly created graph configuration is now visible.

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 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 \( \text{http://dbpedia.org/resource/Nvidia} \):

And for \( \text{http://dbpedia.org/resource/BMW} \):
5.16.3.2 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 and an Ontop one. We will divide a dataset between them and then explore the relationships.

We will be using 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 RDF-ized using Ontotext Refine.

- The Ontop repository uses the prices.csv file of the NYSE dataset, a data source listing the opening/closing stock price and traded volumes on the New York Stock Exchange. The file lists stock symbols and opening/closing stock market prices for particular dates. Most data span from 2010 to the end 2016, and 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 FedEx repository where these two will be federated:
   a. Create it via Setup Repositories Create new repository FedEx Virtual SPARQL.
b. In the configuration screen, include the two repositories that we created as members by clicking on them.

Create FedEx Virtual SPARQL repository

<table>
<thead>
<tr>
<th>Location</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository ID*</td>
<td>fed-repo-native-investments</td>
</tr>
<tr>
<td>Repository description</td>
<td></td>
</tr>
<tr>
<td>Federation members</td>
<td></td>
</tr>
<tr>
<td>No members - add at least one local or remote repository</td>
<td></td>
</tr>
</tbody>
</table>

Create Fedora Virtual SPARQL repository

<table>
<thead>
<tr>
<th>Location</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository ID*</td>
<td>fed-repo-native-investments</td>
</tr>
<tr>
<td>Repository description</td>
<td></td>
</tr>
<tr>
<td>Federation members</td>
<td></td>
</tr>
<tr>
<td>No members - add at least one local or remote repository</td>
<td></td>
</tr>
</tbody>
</table>

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 {
  ?buyerCompany cb:country "USA" .
}
```

The first two triples represent the acquiring and the acquired company. The “USA” literal specifies that the buyer company is based there. The target company has to be European. The country of each company is represented by a country code. To get only the European companies that have been acquired, a filter is used that checks if a given country’s code is among the listed ones.

The first 15 returned results look like this:
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 of the buyer company.

We also make sure 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.

```sparql
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 {
  ?buyerCompany cb:country "USA" .
  ?id cb:acquiredAt ?date .
  ?dayTrade ny:date ?date .
  ?dayTrade ny:close ?close .
}
```
The first 15 returned results look like this:

<table>
<thead>
<tr>
<th>sellingCompany</th>
<th>buyerCompany</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptMedica</td>
<td>Abbott Labs</td>
</tr>
<tr>
<td>IDEV technologies</td>
<td>Abbott Labs</td>
</tr>
<tr>
<td>Facet Biotech</td>
<td>Abbott Labs</td>
</tr>
<tr>
<td>Livia Radio</td>
<td>Ford</td>
</tr>
<tr>
<td>Aragon Pharmaceuticals</td>
<td>Johnson &amp; Johnson</td>
</tr>
<tr>
<td>Calibra Medical</td>
<td>Johnson &amp; Johnson</td>
</tr>
<tr>
<td>MobileAccess Networks</td>
<td>Corning</td>
</tr>
<tr>
<td>MoData</td>
<td>Microsoft</td>
</tr>
<tr>
<td>StarSimple</td>
<td>Microsoft</td>
</tr>
<tr>
<td>PhoneFactor</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Skype</td>
<td>Microsoft</td>
</tr>
<tr>
<td>MetricsHub</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Appihorny</td>
<td>Microsoft</td>
</tr>
<tr>
<td>4Wcode</td>
<td>Microsoft</td>
</tr>
<tr>
<td>RADIL</td>
<td>IDEXX Laboratories</td>
</tr>
</tbody>
</table>

5.16.3.3 FedEx with enabled GraphDB security

When creating a FedEx repository with local members, we can specify whether the FedEx repo 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 SPARQL editor of factforge.net and execute:

   ```sparql
   CONSTRUCT WHERE {
   } 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 FedEx 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 {
}
```

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):

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td><a href="http://dbpedia.org/resource/UI_Theater_Dr_Pauls">http://dbpedia.org/resource/UI_Theater_Dr_Pauls</a></td>
<td><a href="http://www.w3.org/2002/01/rdf-schema#seeAlso">http://www.w3.org/2002/01/rdf-schema#seeAlso</a></td>
<td><a href="http://dbpedia.org/resource/London">http://dbpedia.org/resource/London</a></td>
<td></td>
</tr>
</tbody>
</table>

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 FedEx repository to enter 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:

```sparql
SELECT * WHERE {
}
```

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 FedEx repository is instructed to respect the rights for it.
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 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 {
}
```

16. We will see that the returned results (193 in total, like in the above example without security enabled) 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 default admin user, i.e., username: “admin”, password: “root”.

3. Go to the SPARQL editor of [factforge.net](http://factforge.net) and execute:

```
CONSTRUCT WHERE {
} 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 member the “sofia-london” we just created:
   - Select the **GraphDB/RDF4J server** option.
   - As **Server URL**, enter the URL of the remote repository - `http://localhost:7201/`.
   - **Repository ID** is the name of the remote repo - `remote-repo-paris`.
   - **Authentication credentials** are the user and password for the remote repo.
   - **Add**.
7. Restart the repository.
8. In the SPARQL editor, execute:

```
SELECT * WHERE {
}
```

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

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 default admin user, i.e., username: “admin”, password: “root”.
   - Import the Paris data in it.

2. In the first GraphDB instance on port 7200, attach “remote-repo-paris” as a remote location following these steps. For Authentication type, select Basic auth, and input the credentials.
3. Again in the 7200 GraphDB instance, open the edit view of the “fedx-sofia-london” repository.

4. In it, include as 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:

   ```sparql
   SELECT * WHERE {
   }
   ```

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

**Hint:** You can configure *signature authentication* for remote endpoints in the same way.
5.16.4 Configuration parameters

When configuring a FedX repository, several configuration options (described in detail below) 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/disables query logging. Prints the query in the logs. Default is false.
- **Log query plan**: Enables/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, i.e., 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)
5.16.5 Limitations

Some limitations of the current implementation of the GraphDB FedEx federation are:

- DESCRIPT queries are not supported.
- FedEx is not stable with queries of the type `{?s ?p ?o} UNION {?s ?p1 ?o} FILTER (xxx)
- Currently, the federation only works with remote repositories, i.e., 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.

5.17 GraphDB Dev Guide

5.17.1 Storage

5.17.1.1 What is GraphDB’s persistence strategy

GraphDB stores all of its data (statements, indexes, entity pool, etc.) 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. There are many other additional data structures that are used to enable the efficient manipulation of RDF data, but these are not listed, since these internal mechanisms cannot be configured.

5.17.1.2 GraphDB’s indexing options

There are indexing options that offer considerable advantages for specific datasets, retrieval patterns and query loads. Most of them are disabled by default, so you need to enable them as necessary.

Note: Unless stated otherwise, GraphDB allows you to switch indexes on and off against an already populated repository. The repository has to be shut down before the change of the configuration is specified. The next time the repository is started, GraphDB will create or remove the corresponding index. If the repository is already loaded with a large volume of data, switching on a new index can lead to considerable delays during initialization – this is the time required for building the new index.
Transaction control

Transaction support is exposed via RDF4J’s `RepositoryConnection` interface. The three methods of this interface that give you control when updates are committed to the repository are as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void begin()</code></td>
<td>Begins a transaction. Subsequent changes effected through update operations will only become permanent after <code>commit()</code> is called.</td>
</tr>
<tr>
<td><code>void commit()</code></td>
<td>Commits all updates that have been performed through this connection since the last call to <code>begin()</code>.</td>
</tr>
<tr>
<td><code>void rollback()</code></td>
<td>Rolls back all updates that have been performed through this connection since the last call to <code>begin()</code>.</td>
</tr>
</tbody>
</table>

GraphDB supports the so called ‘read committed’ transaction isolation level, which is well-known to relational database management systems - i.e., pending updates are not visible to other connected users, until the complete update transaction has been committed. It guarantees that changes will not impact 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, i.e., GraphDB processes the commits one after another;
- Update transactions do not block read requests in any way, i.e., hundreds of SPARQL queries can be evaluated in parallel (the processing is properly multi-threaded) while update transactions are being handled on separate threads.
- Multiple update/modification/write transactions cannot be initiated and stay open simultaneously, i.e., when a transaction is initiated and started to modify the underlying indexes, no other transaction must be allowed to change anything until the first one is either committed or rollbacked;

**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.

**Note:** In GraphDB SE, the result of leading update operations in a transaction is visible to trailing ones. Due to a limitation of the cluster protocol, this feature is not supported in GraphDB cluster, i.e., an uncommitted transaction will not affect the ‘view’ of the repository through any connection, including the connection used to do the modification.

**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’, i.e.:

- subject-predicate (SP)
- object-predicate (OP)

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 a very large number (greater than around 1,000) of different predicates.
**Context index**

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

**Literal index**

GraphDB automatically builds a literal index allowing faster look-ups of numeric and date/time object values. The index is used during query evaluation only if a query or a subquery (e.g., UNION) has a filter that is comprised of a conjunction of literal constraints using comparisons and equality (not negation or inequality), e.g., \( \text{FILTER(} ?x = 100 \&\& ?y <= 5 \&\& ?\text{start} > "2001-01-01"^\text{xsd:date}) \). Other patterns will not use the index, i.e., 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"^\text{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 decision of the query optimizer whether to make use of this index is statistics-based. If the estimated number of matches for a filter constraint is large relative to the rest of the query, e.g., a constraint with 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 (i.e., above approximately 1e309 and below -1e309) will not work properly.

**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, however 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 API

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

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

The method retrieves statements by ‘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 examples show how to retrieve only implicit statements:

```java
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
}
```

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.

**Tip:** To learn more, see *Query behavior*.

5.17.2 Notifications

5.17.2.1 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 `NotifyingRepositoryConnection`. However, the GraphDB notifications API works at a lower level and uses the internal raw entity IDs for subject, predicate and object instead of Java objects. The benefit of this is that a much higher performance is possible. The downside is that the client must do a separate lookup to get the actual entity values and because of this, the notification mechanism works only when the client is running inside the same JVM as the repository instance.
How to register for local notifications

To receive notifications, register by providing a SPARQL query.

**Note:** The SPARQL query is interpreted as a plain graph pattern by ignoring all more complicated SPARQL constructs such as `FILTER`, `OPTIONAL`, `DISTINCT`, `LIMIT`, `ORDER BY`, etc. 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:

```java
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 matching the pattern `?s rdf:type ?o`.

**Note:** In general, notifications are sent for all incoming triples, which contribute to a solution of the query. The integer parameters in the `notifyMatch` method can be mapped to values using the `EntityPool` object. Furthermore, any statements inferred from newly inserted statements are also subject to handling by the notification mechanism, i.e., clients are notified also 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 not even bound to the chronological order of the statements insertion 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.

### 5.17.2.2 What are GraphDB remote notifications

GraphDB’s remote notification mechanism provides filtered statement add/remove and transaction begin/end notifications for a local or a remote GraphDB repository. Subscribers for this mechanism use patterns of subject, predicate and object (with wildcards) to filter the statement notifications. JMX is used internally as a transport mechanism.
How to use remote notifications

To register/deregister for notifications, use the `NotifyingOwlimConnection` class, which is located in the `graphdb-notifications-<version>.jar` in the `lib` folder of the distribution .zip file. This class wraps a `RepositoryConnection` object connected to a GraphDB repository and provides an API to add/remove notification listeners of the type `RepositoryNotificationsListener`.

Here is a simple example of how to use the API when the GraphDB repository is initialized in the same JVM that runs the example (local repository):

```java
RepositoryConnection conn = null;
// initialize repository connection to GraphDB ...
RepositoryNotificationsListener listener = new RepositoryNotificationsListener() {
    @Override
    public void addStatement(Resource subject, URI predicate,
                               Value object, Resource context,
                               boolean isExplicit, long tid) {
        System.out.println("Added: " + subject + " " + predicate + " " + object);
    }
    @Override
    public void removeStatement(Resource subject, URI predicate,
                                 Value object, Resource context,
                                 boolean isExplicit, long tid) {
        System.out.println("Removed: " + subject + " " + predicate + " " + object);
    }
    @Override
    public void transactionStarted(long tid) {
        System.out.println("Started transaction " + tid);
    }
    @Override
    public void transactionComplete(long tid) {
        System.out.println("Finished transaction " + tid);
    }
};
NotifyingOwlimConnection nConn = new NotifyingOwlimConnection(conn);
IRI ex = SimpleValueFactory.getInstance().createIRI("http://example.com/");
// subscribe for statements with 'ex' as subject
nConn.subscribe(listener, ex, null, null);
// note that this could be any other connection to the same repository
conn.add(ex, ex, ex);
conn.commit();
// statement added should have been printed out
nConn.unsubscribe(listener);

Note: The `transactionStarted()` and `transactionComplete()` events are not bound to any statement. They are dispatched to all subscribers, no matter what they are subscribed for. This means that pairs of start/complete events can be detected by the client without receiving any statement notifications in between.

To use a remote repository (e.g., `HTTPRepository`), the notifying repository connection should be initialized differently:

```java
NotifyingOwlimConnection nConn =
    new NotifyingOwlimConnection(conn, host, port);
```

where `host` (String) and `port` (int) are the host name of the remote machine, in which the repository resides and the port number of the JMX service in the repository JVM. The other part of the above example is also valid for a
remote repository.
The repository connection used to initialize a `NotifyingOwlimConnection` instance can be a `ReplicationClusterConnection`, in which case the notifications work with a GraphDB EE master node (transparently to the user). No changes on the client side are required.

**How to configure remote notifications**

For remote notifications, where the subscriber and the repository are running in different JVM instances (possibly on different hosts), a JMX remote service should be configured in the repository JVM.

This is done by adding the following parameters to the JVM command line:

```
-Dcom.sun.management.jmxremote.port=1717
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

If the repository is running inside a servlet container, these parameters must be passed to the JVM that runs the container and GraphDB. For Tomcat, this can be done using the `JAVA_OPTS` or `CATALINA_OPTS` environment variable.

The port number used should be exactly the port number that is passed to the `NotifyingOwlimConnection` constructor (as in the example above). You have to make sure that the specified port (e.g., 1717) is accessible remotely, i.e., no firewalls or NAT redirection prevent access to it.

In a GraphDB EE cluster setup, all worker nodes need to have their JMX configured properly in order to enable notifications for the whole cluster. The master node assumes that each worker exposes its JMX service on port 1717, but this can be overridden when nodes are added to the cluster (the third parameter to `addClusterNode()` operation is the JMX service port of that node) or by editing the `cluster.properties` configuration file and adding the following parameter:

```
jmxport<N> = <PORTN>
```

where N is the consecutive number of the node you want to configure and PORTN is the port number of that node’s JMX service. Cluster workers also need to have their `com.sun.management.jmxremote.*` JVM parameters properly configured. The GraphDB EE cluster master nodes will therefore be controlled and emit notifications using the same JMX port number.

### 5.17.3 Query behavior

#### 5.17.3.1 What are named graphs

**Hint:** 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:

<table>
<thead>
<tr>
<th>Query</th>
<th>Bindings</th>
</tr>
</thead>
</table>

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 } }`.

### 5.17.3.2 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.

### 5.17.3.3 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.

5.17.3.4 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).

```java
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.:

```java
TupleQuery query = connection.prepareStatement(QueryLanguage.SPARQL, queryString);
query.setDataset(dataset);
```

5.17.3.5 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 {
} 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 {
  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" .
} 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 {
} order by ?o```

- As above, but using an `xsd:long` datatype for the constant within a `FILTER` condition:

```select * where {
  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 {
}

• 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 {
  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 {
  FILTER (str(<http://www.ontotext.com/owlim/entity#id>(?s)) = "115").
}
```

5.17.3.6 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#>
```

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Definition</th>
</tr>
</thead>
</table>
| 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. |
5.17.3.7 Other special GraphDB query behavior

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.

5.17.4 Retain BIND position special graph

The default behavior of the GraphDB query optimizer is to try and reposition BIND clauses so that all the variables within its EXPR part (on the left side of ‘AS’) are bound to have valid bindings for all of the variables referred within it.

If you look at the following data:

```ini
INSERT DATA {
    <urn:q> <urn:pp1> 1 .
    <urn:q> <urn:pp2> 2 .
    <urn:q> <urn:pp3> 3 .
}
```

and try to evaluate a SPARQL query such as the one below (without any rearrangement of the statement patterns):

```sparql
SELECT ?r {
    ?q <urn:pp1> ?x .
    BIND (?x + ?y + ?z AS ?r) .
    ?q <urn:pp3> ?z .
}
```

the ‘correct’ result would be:
because the expression that sums several variables will not produce any valid bindings for ?r.

But if you rearrange the statement patterns in the same query so that you have bindings for all of the variables used within the sum expression of the BIND clause:

```
SELECT ?r {
  ?q <urn:pp1> ?x .
  ?q <urn:pp3> ?z .
  BIND (?x + ?y + ?z AS ?r) .
}
```

the query would return a single result and now the value bound to ?r will be 6:

```
1 result: r=6
```

By default, the GraphDB query optimizer tries to move the BIND after the last statement pattern, so that all the variables referred internally have a binding. However, that behavior can be modified by using a special ‘system’ graph within the dataset section of the query (e.g., as FROM clause) that has the following URI:

```html
<http://www.ontotext.com/retain-bind-position>
```

In this case, the optimizer retains the relative position of the BIND operator within the group in which it appears, so that if you evaluate the following query against the GraphDB repository:

```
SELECT ?r
FROM <http://www.ontotext.com/retain-bind-position> {
  ?q <urn:pp1> ?x .
  BIND (?x + ?y + ?z AS ?r) .
  ?q <urn:pp3> ?z .
}
```

you will get the following result:

```
1 result: r=UNDEF
```

Still, the default evaluation without the special ‘system’ graph provides a more useful result:

```
1 result: r="6"
```

### 5.17.5 Graph replacement optimization

Clearing and old graph and then importing the new information can often be inefficient. Since the two operations are handled separately, it is impossible to determine if a statement will also be present in the new graph and therefore, keep it there. The same applies for preserving connectors or inferring statements. Therefore, GraphDB offers an optimized graph replacement algorithm, making graph updates faster in those situations where the new graph will partially overlap with data in the old one.

The graph replacement optimization is in effect when the replacement is done in a single transaction and when the transaction is bigger than a certain threshold. By default, this threshold is set to 1,000, but it can be controlled by using the `graphdb.engine.min-replace-graph-tx-size` configuration parameter.

The algorithm has the following steps:

1. Check transaction contents. If the transaction includes a graph replacement and is of sufficient size, 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:

```bash
curl -X PUT -H "Content-Type: application/x-trig" --data-binary '@test_modified.trig'
```

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

```bash
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. i.e., 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://www.ontotext.com/replaceGraph> <http://example.org/graph>

### 5.18 Experimental Features

#### 5.18.1 SPARQL-MM support

SPARQL-MM is a multimedia extension for SPARQL 1.1. The implementation is based on code developed by Thomas Kurz, and is implemented as a GraphDB plugin.

**5.18.1.1 Usage examples**

**Temporal relations**

```sparql
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 {
  mm:precedes(?f1,?f2)
} ORDER BY ?t1 ?t2
```
Temporal aggregation

```sql
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

```sql
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

```sql
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 {
  FILTER mm:rightBeside(?f1,?f2)
} ORDER BY ?t1 ?t2
```

Spatial aggregation

```sql
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

```sql
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 {
  FILTER mm:equals(?f1,?f2)
} ORDER BY ?t1 ?t2
```
General aggregation

```sql
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

```sql
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
```

**Tip:** For more information, see:

- The SPARQL-MM Specification
- List of SPARQL-MM functions
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:

### 6.1 Access Control

#### 6.1.1 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.

##### 6.1.1.1 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 RBAC1 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

<table>
<thead>
<tr>
<th>Role name</th>
<th>Inherits roles</th>
<th>Associated permissions (without the inherited ones)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE_ADMIN</td>
<td>ROLE_REPO_MANAGER ROLE_CLUSTER</td>
<td>Can perform all operations, i.e., the security never rejects an operation</td>
</tr>
<tr>
<td>ROLE_REPO_MANAGER</td>
<td></td>
<td>Can create, edit, and delete repositories with read and write permissions to all repositories</td>
</tr>
<tr>
<td>ROLE_MONITORING</td>
<td>ROLE_USER</td>
<td>Allows monitoring operations (queries, updates, abort query/update, resource monitoring)</td>
</tr>
<tr>
<td>ROLE_USER</td>
<td></td>
<td>Can save SPARQL queries, graph visualizations, or user-specific settings</td>
</tr>
<tr>
<td>ROLE_CLUSTER</td>
<td></td>
<td>Can perform internal cluster operations</td>
</tr>
</tbody>
</table>

The following repository access roles are available as well:

<table>
<thead>
<tr>
<th>Role name</th>
<th>Associated permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_REPO_*</td>
<td>Read permissions to all repository IDs</td>
</tr>
<tr>
<td>WRITE_REPO_*</td>
<td>Write permissions to all repository IDs</td>
</tr>
<tr>
<td>READ_REPO_xxx</td>
<td>Read permissions to a given repository, where xxx is the repository ID</td>
</tr>
<tr>
<td>WRITE_REPO_xxx</td>
<td>Write permissions to a given repository, where xxx is the repository ID</td>
</tr>
</tbody>
</table>

**Note:** When providing the WRITE_REPO_xxx role for a given repository, the READ_REPO_xxx role must be provided for it as well.

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:

<table>
<thead>
<tr>
<th>Inherent role and permissions</th>
<th>Regular user</th>
<th>Repository manager</th>
<th>Administrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core role</td>
<td>ROLE_USER</td>
<td>ROLE_REPO_MANAGER</td>
<td>ROLE_ADMIN</td>
</tr>
<tr>
<td>Read access to a specific repository</td>
<td>optional</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Read/write access to a specific repository</td>
<td>optional</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Read/write access to all repositories</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Create, edit, and delete repositories</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Access monitoring</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Manage Connectors</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Manage Users and Access</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Manage the cluster</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Attach remote locations</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>View system information</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
6.1.1.2 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.

<table>
<thead>
<tr>
<th>Username</th>
<th>Associated roles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cluster user&gt;</td>
<td>ROLE_CLUSTER</td>
<td>Used for cluster-internal communication between cluster nodes.</td>
</tr>
<tr>
<td></td>
<td>READ_REPO_*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE_REPO_*</td>
<td></td>
</tr>
<tr>
<td>&lt;free access user&gt;</td>
<td>None by default but administrators may grant access to one or more repositories.</td>
<td>The user associated with anonymous access if anonymous access is enabled. See how to configure free access here.</td>
</tr>
</tbody>
</table>

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.

6.1.1.3 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:

```javascript
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.

![Settings](image)

**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.

### 6.1.1.4 LDAP user database

**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:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphdb.auth.ldap.url</code> <em>(required)</em></td>
<td>LDAP endpoint</td>
<td><code>ldap://&lt;my-openldap-server&gt;:389/&lt;partition&gt;</code></td>
</tr>
<tr>
<td><code>graphdb.auth.ldap.user.search.base</code></td>
<td>Query to identify the directory where all authenticated users are located.</td>
<td><code>&lt;empty&gt;</code></td>
</tr>
<tr>
<td><code>graphdb.auth.ldap.user.search.filter *(required)*</code></td>
<td>Matches the attribute to a GraphDB username.</td>
<td><code>({0})</code></td>
</tr>
<tr>
<td><code>graphdb.auth.ldap.role.search.base *(required)*</code></td>
<td>Query to identify the directory where roles/groups for authenticated users are located.</td>
<td><code>&lt;empty&gt;</code></td>
</tr>
</tbody>
</table>
| `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:  
  - `{0}` is replaced with the full LDAP distinguished name, e.g., `cn=johnsmith, o=example, o=com`.  
  - `{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}`. | `(uniqueMember={1})` |
| `graphdb.auth.ldap.role.map.administrator` *(required)* | Map a single LDAP group to GDB administrator role. | `my-group-name` |
| `graphdb.auth.ldap.role.map.repositoryManager` | Map a single LDAP group to GDB repository manager role. | `my-group-name` |
| `graphdb.auth.ldap.role.map.repository.read.<my-repo>` | Map a single LDAP group to GDB repository-specific read permissions. | `my-group-name` |
| `graphdb.auth.ldap.role.map.repository.write.<my-repo>` | Map a single LDAP group to GDB repository-specific write permissions. | `my-group-name` |
| `graphdb.auth.ldap.repository.read.base` | Query to identify the directory where repository read groups for authenticated users are located. | |
| `graphdb.auth.ldap.repository.read.filter` | Authorize a user by matching the manner in which they are listed within the group. | `(uniqueMember={0})` |
| `graphdb.auth.ldap.repository.read.attribute` | The attribute to identify the common name. | `cn` *(default)* |
| `graphdb.auth.ldap.repository.write.base` | Query to identify the directory where repository write groups for authenticated users are located. | |
| `graphdb.auth.ldap.repository.write.filter` | Authorize a user by matching the manner in which they are listed within the group. | `(uniqueMember={0})` |
| `graphdb.auth.ldap.repository.write.attribute` | The attribute to identify the common name. | `cn` *(default)* |

**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:

```xml
<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.
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:

```
grapdb.auth.ldap.role.map.administrator = gdbadmin
```

Mapping the Repository manager role

Set the following property to the LDAP group that must receive this role:

```
grapdb.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.

6.1.1.5 OAuth user database

**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:

```
grapdb.auth.database = oauth
```

When OAuth authorization is enabled, the following property settings can be used to configure it:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphdb.auth.oauth.roles_claim</code> (required)</td>
<td>OAuth roles claim. The field from the JWT token that will provide the GraphDB roles. No default value.</td>
<td>roles</td>
</tr>
<tr>
<td><code>graphdb.auth.oauth.default_roles</code> (required)</td>
<td>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.</td>
<td>ROLE_USER</td>
</tr>
<tr>
<td><code>graphdb.auth.oauth.roles_prefix</code></td>
<td>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. The default value is the empty string.</td>
<td>GDB_</td>
</tr>
<tr>
<td><code>graphdb.auth.oauth.roles_suffix</code></td>
<td>OAuth roles suffix to strip. The roles claim may provide the GraphDB roles with some suffix, e.g., GDB_ROLE_USER. The suffix will be stripped when the roles are mapped. The default value is the empty string.</td>
<td>_GDB</td>
</tr>
</tbody>
</table>

**Note:** GraphDB enables 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 an external user database (LDAP, OpenID), the external database must support case-insensitive login as well.

### 6.1.2 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).

• 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).

• Kerberos: highly secure single sign-on protocol that uses tickets for authentication. Disabled by default (must be configured to be enabled).

All four authentication providers - Basic, GDB, OpenID, 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:
### 6.1.2.1 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 Base64 encoded, they will also get your username and password. This is why it is very important to always use encryption in transit.

### 6.1.2.2 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:
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
```

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.

### 6.1.2.3 OpenID authentication

**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:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphdb.auth.openid.issuer</code> <em>(required)</em></td>
<td>OpenID issuer URL used to derive keys, endpoints, and token validation. No default value.</td>
<td><a href="https://accounts.example.com">https://accounts.example.com</a></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.client_id</code> <em>(required)</em></td>
<td>OpenID client ID used to authenticate and validate tokens. No default value.</td>
<td><code>&lt;my-client-id&gt;</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.username_claim</code> <em>(required)</em></td>
<td>OpenID claim to use as the GraphDB username. No default value.</td>
<td><code>email</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.auth_flow</code> <em>(required)</em></td>
<td>OpenID authentication flow: code, code_no_pkce, implicit. The recommended value is <code>code</code> if the OpenID provider supports it with PKCE without a client secret. No default value.</td>
<td><code>code</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.token_type</code> <em>(required)</em></td>
<td>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.</td>
<td><code>access</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.issuer</code></td>
<td>OpenID expected issuer URL in tokens, used to validate tokens. The default is the same as the actual issuer URL.</td>
<td><a href="https://accounts.example.com/custom">https://accounts.example.com/custom</a></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.token_audience</code></td>
<td>OpenID expected audience in tokens, used to validate tokens. The default is the same as the client ID.</td>
<td><code>&lt;my-audience&gt;</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.authorize_parameters</code></td>
<td>OpenID extra parameters for the authorize endpoint. Some OpenID providers require additional parameters sent to the authorize endpoint (e.g., <code>resource=xxx</code>). This is a URL-encoded string where each parameter-value pair is delimited by &amp;. The string will be appended to the rest of the authorize URL parameters. The default value is the empty string.</td>
<td><code>param1=value%201&amp;param2=value%202</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.proxy</code></td>
<td>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 <code>true/false</code>. False by default.</td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>graphdb.auth.openid.extra_scopes</code></td>
<td>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 supported, the ‘offline_access’ scope. Scopes are used to request sets of claims, e.g., you might need to set this to a provider-specific value in order to obtain the username_claim or the roles_claim (if using OAuth as well). The default value is profile email.</td>
<td></td>
</tr>
</tbody>
</table>
**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.

### 6.1.2.4 Kerberos authentication

**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 [here](#).

Enable Kerberos with the following property:

```
graphdb.auth.methods = basic, gdb, kerberos
```

The default value is `basic, gdb`.

Kerberos is configured via several properties:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphdb.auth.kerberos.keytab (required)</td>
<td>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.</td>
<td>graphdb-http.keytab</td>
</tr>
<tr>
<td>graphdb.auth.kerberos.principal (required)</td>
<td>Name of the Kerberos service principal. Required if Kerberos is enabled.</td>
<td>HTTP/data.example.com@EXAMPLE.COM</td>
</tr>
<tr>
<td>graphdb.auth.kerberos.debug</td>
<td>Whether some of the Spring Kerberos classes print extra messages related to Kerberos.</td>
<td>false</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Service principal</th>
<th>Principal from SPNEGO token</th>
<th>Username in GraphDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/data.example.com@EXAMPLE.COM</td>
<td><a href="mailto:john@EXAMPLE.COM">john@EXAMPLE.COM</a></td>
<td>john</td>
</tr>
<tr>
<td>HTTP/data.example.com@EXAMPLE.COM</td>
<td><a href="mailto:john@FOO.EXAMPLE.COM">john@FOO.EXAMPLE.COM</a></td>
<td>Invalid authentication because of realm mismatch</td>
</tr>
</tbody>
</table>

**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.

6.1.3 Example configurations

This is a list of example configurations for some of the possible combinations of authentication methods (Basic, GDB, OpenID, and Kerberos) with the three supported user databases for authorization (Local, LDAP, and OAuth).

Hint: The OpenID, Kerberos and LDAP part of the examples is identical in all cases but is repeated for convenience.

6.1.3.1 Basic/GDB + LDAP

Example configuration of Basic and GDB authentication + LDAP authorization:

```java
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LDAP AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on ldap authentication and configure the server.
graphdb.auth.database = 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
```
6.1.3.2 OpenID + Local users

Example configuration of OpenID authentication + local user database authorization:

```plaintext
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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
```

6.1.3.3 OpenID + LDAP

Example configuration for OpenID authentication + LDAP authorization:

```plaintext
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ LDAP AUTHORIZATION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Turn on ldap authentication and configure the server.
graphdb.auth.module = ldap
```

(continues on next page)
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

6.1.3.4 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.
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_
### 6.1.3.5 Kerberos + Local users

Example configuration for Kerberos authentication + local user database authorization:

```yaml
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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
```

### 6.1.3.6 Kerberos + LDAP

Example configuration for Kerberos authentication + LDAP authorization:

```yaml
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ 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})
```

(continues on next page)
# 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 Encryption

6.2.1 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.

6.2.1.1 Configuring 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 three parameters:

- graphdb.connector.SSLEnabled = true
- graphdb.connector.scheme = https
- graphdb.connector.secure = true

GraphDB uses the Java implementation of SSL, which requires a configured key in the Java keystore.

- graphdb.connector.keystoreFile = <path to the keystore file>
- graphdb.connector.keystorePass = <secret>
- graphdb.connector.keyAlias = graphdb
- graphdb.connector.keyPass = <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.

### 6.2.1.2 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
6.2.2 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.

6.3 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.

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.
7.1 Data Modeling with RDF(S)

7.1.1 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:

<table>
<thead>
<tr>
<th>subject</th>
<th>predicate</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Fred</td>
<td>:hasSpouse</td>
<td>:Wilma</td>
</tr>
<tr>
<td>:Fred</td>
<td>:hasAge</td>
<td>25</td>
</tr>
</tbody>
</table>

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.

### 7.1.2 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.

### 7.2 SPARQL

#### 7.2.1 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:
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:

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:

```sparql
PREFIX: <http://bedrock/>
SELECT ?grandChild
WHERE {
  :fred :hasChild ?child .
}
```

The query results, reflected in the red box, tell us that Fred’s grandchildren are Roxy and Chip.

### 7.2.2 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:

#### SPARQL Query & Update

<table>
<thead>
<tr>
<th>Unnamed</th>
<th>Editor only</th>
<th>Editor and results</th>
<th>Results only</th>
</tr>
</thead>
</table>
| 1. select * where {  
  } limit 100  
|          |             |                    |              |

The query results, reflected in the red box, tell us that Fred’s grandchildren are Roxy and Chip.

#### 7.3 RDF-star and SPARQL-star

**7.3.1 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.
7.3.1.1 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:

```xml
@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.

7.3.1.2 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:

```xml
@prefix voc: <http://example.com/voc#> .

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.

7.3.1.3 Singleton properties

Singleton properties are a hacky way to introduce statement identifiers as a part of the predicate like:

```xml
@prefix voc: <http://example.com/voc#> .

voc:man voc:hasSpouse#1 voc:woman .
voc:hasSpouse#1 voc:startDate "2020-02-11"^^xsd:date .
```

The local name of the predicate after the # encodes a unique identifier. The approach is compact for exchanging data since it uses only two statements, but is highly inefficient for querying data. A query to return all :hasSpouse links must parse all predicate values with a regular expression.

**Warning:** GraphDB supports singleton properties in a reasonably inefficient way. The database expects the number of unique predicates to be much smaller than the total number of statements. Our recommendation is to avoid this modeling approach for models with significant size.
7.3.1.4 Named graphs

The named graph approach is a variation of the singleton properties, where a unique value on the named graph position identifies the statement like:

```prefix
@prefix voc: <http://example.com/voc#> .

voc:man voc:hasSpouse voc:woman voc:statementId#1 .
voc:statementId#1 voc:startDate "2020-02-11"^^xsd:date voc:metadata .
```

The approach has multiple advantages over the singleton properties and eliminates the need for regular expression parsing. A significant drawback is the overload of the named graph parameter with an identifier instead of the file or source that produced the triple. The updates based on the triple source become more complicated and cumbersome to maintain.

**Tip:** If a repository stores a large number of named graphs, make sure to enable the context indexes.

7.3.1.5 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
@prefix voc: <http://example.com/voc#> .

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
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
SELECT *
WHERE {
    <<?man voc:hasSpouse voc:woman>> ?p ?o
    FILTER (?man = voc:man)
}
```
7.3.2 How 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:

<table>
<thead>
<tr>
<th>Modeling approach</th>
<th>Total statements</th>
<th>Loading time (min)</th>
<th>Repository image size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard reification</td>
<td>391,652,270</td>
<td>52.4</td>
<td>36,768</td>
</tr>
<tr>
<td>N-ary relations</td>
<td>334,571,877</td>
<td>50.6</td>
<td>34,519</td>
</tr>
<tr>
<td>Named graphs</td>
<td>277,478,521</td>
<td>56</td>
<td>35,146</td>
</tr>
<tr>
<td>RDF-star</td>
<td>220,375,702</td>
<td>34</td>
<td>22,465</td>
</tr>
</tbody>
</table>

We did not test the singleton properties approach due to the high number of unique predicates.

7.3.3 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:

```expect
@prefix voc: <http://example.com/voc#> .

<<voc:man voc:hasSpouse :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 utilized.

- **Object relation qualifiers**:

  ```expect
  @prefix voc: <http://example.com/voc#> .
  
  <<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:
  
  ```expect
  @prefix voc: <http://example.com/voc#> .
  
  <<voc:woman voc:hasSpouse voc:man> voc:startDate "2020-02-11"^^xsd:date
  ```

- **Data value qualifiers**:

  ```expect
  @prefix voc: <http://example.com/voc#> .
  
  <<voc:painting voc:height 32.1>>
  voc:unit voc:cm;
  ```

(continues on next page)
• Statement sources/references:

```sparql
@prefix voc: <http://example.com/voc#> .
<<voc:man voc:hasSpouse voc:woman>>
  voc:source voc:TheNationalEnquirer;
  voc:retrieved "2020-02-13"^^xsd:dateTime.
```

• Nested embedded triples:

```sparql
@prefix voc: <http://example.com/voc#> .
<< <<voc:man voc:hasSpouse voc:woman>> voc:startDate "2020-02-11"^^xsd:date >>
```

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:

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?p ?a ?c WHERE {
}
```

Additionally, SPARQL-star modifies the `BIND` clauses to select a group of embedded triples by using free variables:

```sparql
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)
  }
}
```

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.
# Values generates new values

```sparql
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:

```sparql
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)
}
```

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.

### 7.3.3.1 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 GraphDB 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.

7.3. RDF-star and SPARQL-star
7.3.4 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:

```ttl
@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 .

<<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:

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
DELETE {
  ?reification a rdf:Statement .
}
INSERT {
}
WHERE {
  ?reification a rdf:Statement .
  FILTER (?p NOT IN (rdf:subject, rdf:predicate, rdf:object) &&
               (?p != rdf:type && ?object != rdf:Statement))
}
```

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:
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:

<table>
<thead>
<tr>
<th>RDF-star format</th>
<th>MIME type</th>
<th>File extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary RDF</td>
<td>application/x-binary-rdf</td>
<td>brf</td>
</tr>
<tr>
<td>Turtle-star</td>
<td>text/x-turtlestar</td>
<td>ttls</td>
</tr>
<tr>
<td></td>
<td>application/x-turtlestar</td>
<td></td>
</tr>
<tr>
<td>TriG-star</td>
<td>application/x-trigstar</td>
<td>trigs</td>
</tr>
<tr>
<td>JSON-star query result</td>
<td>application/x-sparqlstar-results+json</td>
<td>srjs</td>
</tr>
<tr>
<td>TSV-star query result</td>
<td>text/x-tab-separated-values-star</td>
<td>tsvs</td>
</tr>
<tr>
<td></td>
<td>application/x-sparqlstar-results+tsv</td>
<td></td>
</tr>
<tr>
<td>XML-star query result</td>
<td>application/x-sparqlstar-results+xml</td>
<td>xmls</td>
</tr>
</tbody>
</table>

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 Ontologies

#### 7.4.1 What is an ontology?

An ontology is a formal specification that provides sharable 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`.

7.4.2 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.
7.4.3 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:

![Import view](image)

7.5 Inference

7.5.1 What is inference?

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.

7.5.2 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. The implementation of 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.
7.5.2.1 Standard rulesets

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

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

![Create Repository](image)

7.5.2.2 Custom rulesets

GraphDB also comes with support for custom rulesets that allow for custom reasoning through the same performance optimised inference engine. The rulesets are defined via `.pie` files.

To load custom rulesets, simply point to the location of your `.pie` file as shown below:

![Create Repository](image)

See how to configure the default inference value setting from the Workbench [here](#).

7.6 Programming with GraphDB

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.
7.6.1 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 here.

7.6.2 Examples

The three examples below can be found under examples/developer-getting-started of the GraphDB distribution.

7.6.2.1 Hello world in GraphDB

The following program opens a connection to a repository, evaluates a SPARQL query and prints the result. The example uses an embedded GraphDB instance but it can easily be modified to connect to a remote repository. See also Embedded GraphDB.

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
```

```java
package com.ontotext.graphdb.example.app.hello;

import com.ontotext.graphdb.example.util.EmbeddedGraphDB;
import org.openrdf.model.Value;
import org.openrdf.query.*;
import org.openrdf.repository.RepositoryConnection;

/**
 * Hello World app for GraphDB
 */
public class HelloWorld {
    public void hello() throws Exception {
        // Open connection to a new temporary repository
        // (ruleset is irrelevant for this example)
        RepositoryConnection connection = EmbeddedGraphDB.openConnectionToTemporaryRepository("rdfs");

        /* Alternative: connect to a remote repository
         
         // Abstract representation of a remote repository accessible over HTTP
         HTTPRepository repository = new HTTPRepository("http://localhost:8080/graphdb/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
```

(continues on next page)
// 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
  BindingsSet 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);
  }

  // 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();
}

7.6.2.2 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.EmbeddedGraphDB;
import com.ontotext.graphdb.example.util.QueryUtil;
import com.ontotext.graphdb.example.util.UpdateUtil;
import org.openrdf.model.URI;
import org.openrdf.model.impl.URIImpl;
import org.openrdf.query.*;
import org.openrdf.query.impl.BindingImpl;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.repository.RepositoryException;
import org.openrdf.rio.RDFFormat;
import org.openrdf.rio.RDFParseException;
import java.io.IOException;

/**
 * An example that illustrates loading of ontologies, data, querying and modifying data.
 */
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 ." +
                "?p1 ?r ?p2 ." +
                "?p1 ?r ?p2 ." +
                "?r rdfs:subPropertyOf family:hasRelative ." +
        // (continues on next page)
"FILTER(?r != family:hasRelative)" +
"",
    new BindingImpl("p1", uriForPerson(person));

while (result.hasNext()) {
    BindingSet bindingSet = result.next();
    URI p1 = (URI) bindingSet.getBinding("p1").getValue();
    URI r = (URI) bindingSet.getBinding("r").getValue();
    URI p2 = (URI) bindingSet.getBinding("p2").getValue();

    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((URI) 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);

    URI childURI = uriForPerson(child);
    URI 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 (" +
(continues on next page)
"<s> family:hasChild <s>" + 
""}, personURI, childURI));

// Committing the transaction persists the changes
collection.commit();
}

private URI uriForPerson(String person) {
    return new URIImpl("http://examples.ontotext.com/family/data#" + person);
}

public static void main(String[] args) throws Exception {
    // Open connection to a new temporary repository
    // (in order to infer grandparents/grandchildren we need the OWL2-RL ruleset)
    RepositoryConnection connection = EmbeddedGraphDB.openConnectionToTemporaryRepository("owl2-rl-→optimized");
    /* Alternative: connect to a remote repository
    // Abstract representation of a remote repository accessible over HTTP
    HTTPRepository repository = new HTTPRepository("http://localhost:8080/graphdb/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 {
        // It is best to close the connection in a finally block
        connection.close();
    }
}
7.6.2.3 Embedded GraphDB

Typically, GraphDB is used as a standalone server running as a Tomcat application. Clients then connect to that server over HTTP to execute queries and modify the data.

It is also possible to run GraphDB as an embedded database and use the RDF4J API directly on it, thus bypassing the need for a separate server or even networking. This is typically used to implement testing or simpler applications.

The following example code illustrates how an embedded GraphDB instance can be created. See the example applications Hello world in GraphDB and Family relations app for how to use it.

```java
package com.ontotext.graphdb.example.util;

import com.ontotext.trree.config.OWLIMSailSchema;
import org.eclipse.rdf4j.common.io.FileUtil;
import org.eclipse.rdf4j.model.Literal;
import org.eclipse.rdf4j.model.Resource;
import org.eclipse.rdf4j.model.IRI;
import org.eclipse.rdf4j.model.impl.SimpleValueFactory;
import org.eclipse.rdf4j.model.impl.TreeModel;
import org.eclipse.rdf4j.model.util.Models;
import org.eclipse.rdf4j.model.vocabulary.RDF;
import org.eclipse.rdf4j.model.vocabulary.RDFS;
import org.eclipse.rdf4j.rdf4j.repository.Repository;
import org.eclipse.rdf4j.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.rdf4j.repository.RepositoryException;
import org.eclipse.rdf4j.rdf4j.repository.base.RepositoryConnectionWrapper;
import org.eclipse.rdf4j.rdf4j.repository.config.RepositoryConfig;
import org.eclipse.rdf4j.rdf4j.repository.config.RepositoryConfigException;
import org.eclipse.rdf4j.rdf4j.repository.config.RepositoryConfigSchema;
import org.eclipse.rdf4j.rdf4j.repository.manager.LocalRepositoryManager;
import org.eclipse.rdf4j.rdf4j.repository.sail.config.SailRepositorySchema;
import org.eclipse.rdf4j.rdf4j.rio.*;
import org.eclipse.rdf4j.rio.helpers.StatementCollector;
import java.io.Closeable;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.Collections;
import java.util.Map;

/**
 * A useful class for creating a local (embedded) GraphDB database (no networking needed).
 */
public class EmbeddedGraphDB implements Closeable {
    private LocalRepositoryManager repositoryManager;

    /**
     * Creates a new embedded instance of GraphDB in the provided directory.
     * @param baseDir a directory where to store repositories
     * @throws RepositoryException
     */
    public EmbeddedGraphDB(String baseDir) throws RepositoryException {
        repositoryManager = new LocalRepositoryManager(new File(baseDir));
        repositoryManager.initialize();
    }

    /**
     * Creates a repository with the given ID.
     */
```

(continues on next page)
* @param repositoryId  a new repository ID
* @throws RDFHandlerException
* @throws RepositoryConfigException
* @throws RDFParseException
* @throws IOException
* @throws GraphUtilException
* @throws RepositoryException
*/

public void createRepository(String repositoryId) throws RDFHandlerException,
   RepositoryConfigException, RDFParseException, IOException, RepositoryException {
    createRepository(repositoryId, null, null);
}

/**
 * Creates a repository with the given ID, label and optional override parameters.
 *
 * @param repositoryId  a new repository ID
 * @param repositoryLabel a repository label, or null if none should be set
 * @param overrides a map of repository creation parameters that override the defaults, or null
 
 */

public void createRepository(String repositoryId, String repositoryLabel, Map<String, String> overrides) throws RDFParseException, IOException, RDFHandlerException,
       RepositoryConfigException, RepositoryException {
    if (repositoryManager.hasRepositoryConfig(repositoryId)) {
      throw new RuntimeException("Repository "+ repositoryId + " already exists.");
    }

    TreeModel graph = new TreeModel();
    InputStream config = EmbeddedGraphDB.class.getResourceAsStream("/repo-defaults.ttl");
    RDFParser rdfParser = Rio.createParser(RDFFormat.TURTLE);
    rdfParser.setRDFHandler(new StatementCollector(graph));
    rdfParser.parse(config, RepositoryConfigSchema.NAMESPACE);
    config.close();

    Resource repositoryNode = Models.subject(graph.filter(null, RDF.TYPE, RepositoryConfigSchema.REPOSITORY)).orElse(null);
    graph.add(repositoryNode, RepositoryConfigSchema.REPOSITORYID,
               SimpleValueFactory.getInstance().createLiteral(repositoryId));

    if (repositoryLabel != null) {
      graph.add(repositoryNode, RDFS.LABEL,
                 SimpleValueFactory.getInstance().createLiteral(repositoryLabel));
    }

    if (overrides != null) {
      Resource configNode = (Resource)Models.object(graph.filter(null, SailRepositorySchema.SAILIMPL).orElse(null));
      for (Map.Entry<String, String> e : overrides.entrySet()) {
        IRI key = SimpleValueFactory.getInstance().createIRI(OWLIMSailSchema.NAMESPACE + e.getKey());
        Literal value = SimpleValueFactory.getInstance().createLiteral(e.getValue());
    }

    (continues on next page)
repositoryManager.addRepositoryConfig(repositoryConfig);
}

public Repository getRepository(String repositoryId) throws RepositoryException, RepositoryConfigException {
    return repositoryManager.getRepository(repositoryId);
}

@Override
public void close() throws IOException {
    repositoryManager.shutDown();
}

/**
 * A convenience method to create a temporary repository and open a connection to it.
 * When the connection is closed all underlying objects (EmbeddedGraphDB and LocalRepositoryManager)
 * will be closed as well. The temporary repository is created in a unique temporary directory
 * that will be deleted when the program terminates.
 * @param ruleset ruleset to use for the repository, e.g. owl-horst-optimized
 * @return a RepositoryConnection to a new temporary repository
 * @throws IOException
 * @throws RepositoryException
 * @throws RDFParseException
 * @throws GraphUtilException
 * @throws RepositoryConfigException
 * @throws RDFHandlerException
 */
public static RepositoryConnection openConnectionToTemporaryRepository(String ruleset) throws IOException, RepositoryException, RDFParseException, RepositoryConfigException, RDFHandlerException {
    // Temporary directory where repository data will be stored.
    // The directory will be deleted when the program terminates.
    File baseDir = FileUtil.createTempDir("graphdb-examples");
    baseDir.deleteOnExit();

    // Create an instance of EmbeddedGraphDB and a single repository in it.
    final EmbeddedGraphDB embeddedGraphDB = new EmbeddedGraphDB(baseDir.getAbsolutePath());
    embeddedGraphDB.createRepository("tmp-repo", null, Collections.singletonMap("ruleset", ruleset));

    // Get the newly created repository and open a connection to it.
    Repository repository = embeddedGraphDB.getRepository("tmp-repo");
    RepositoryConnection connection = repository.getConnection();

    // Wrap the connection in order to close the instance of EmbeddedGraphDB on connection close
    return new RepositoryConnectionWrapper(repository, connection) {
        @Override
        public void close() throws RepositoryException {
            super.close();
            try {
                embeddedGraphDB.close();
            } catch (IOException e) {
                throw new RepositoryException(e);
            }
        }
    };
}
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.

### 7.7 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.

#### 7.7.1 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`.

#### 7.7.2 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:

```javascript
$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:

```javascript
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

```javascript
'graphdb.framework.graphexplore.controllers.paths',
```

Now your controller and page are ready to be filled with content.

**7.7.3 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.

```html
<div class="container-fluid">
  <h1>{{title}}</h1>
  <span popover-template="js/angular/templates/titlePopoverTemplate.html"
    popover-trigger="mouseenter"
    popover-placement="bottom-right"
    popover-append-to-body="true">\n    \n    </span>
  <div core-errors></div>
  <div system-repo-warning></div>
  <div class="alert alert-danger ng-show=repositoryError">
    (continues on next page)
  </div>
</div>
```
The currently selected repository cannot be used for queries due to an error: {{repositoryError}}

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.

```javascript
$scope.getActiveRepository = function () {
  return $repositories.getActiveRepository();
};

$scope.isLoadingLocation = function () {
  return $repositories.isLoadingLocation();
};

$scope.hasActiveLocation = function () {
  return $repositories.hasActiveLocation();
};
```

### 7.7.4 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:

```sparql
PREFIX onto: <http://www.ontotext.com/>
INSERT {
  ?node onto:hasFlightTo ?destination.
} WHERE {
}
```

Now we will search for paths between airports based on the hasFlightTo predicate.

### 7.7.5 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 `startNode` 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.
They need the `getNamespacesPromise` and `getAutocompletePromise` to fetch the Autocomplete data. They should be initialized once the repository has been set in the controller.

```javascript
function initForRepository() {
    if (!$repositories.getActiveRepository()) {
        return;
    }
    $scope.getNamespacesPromise = ClassInstanceDetailsService.getNamespaces($scope.
    ->getActiveRepository());
    $scope.getAutocompletePromise = AutocompleteService.checkAutocompleteStatus();
}
```

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.
Graph Paths

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.

```javascript
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)
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.

```javascript
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];
  })))
```
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
            }
        })
    };
}
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 + ");";
    });
}(continued on next page)
});

var predicate = container.selectAll(".link-wrapper")
  .selectAll("text")
  .selectAll(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);
    // TODO: get language and set it on the label html tag
  })
  .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/messages 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')

(continues on next page)
It obtains the URIs for the nodes from all links, and finds their labels through calls to the \texttt{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 \texttt{draw(graph)} function does the D3 drawing itself using the D3 force layout.
7.7.7 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.

7.7.8 Add status message

Noticing that path finding can take some time, we may want to add a message for the user.

```javascript
$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.
7.8 GraphDB REST API

7.8.1 Location and repository management with the GraphDB REST API

The GraphDB REST API can be used for managing locations and repositories programmatically. It includes connecting to remote GraphDB instances (locations), activating a location, and different ways for creating a repository. This tutorial shows how to use cURL command to perform basic location and repository management through the GraphDB REST API.

7.8.1.1 Prerequisites

- One or optionally two machines with Java.
- One GraphDB instance:
  - Start GraphDB on the first machine.
  
    **Tip:** For more information on deploying GraphDB, please see *Installation*.

- Another GraphDB instance (optional, needed for the *Attaching a remote location* example):
  - Start GraphDB on the second machine.

- The cURL command line tool for sending requests to the API.

  **Hint:** Throughout the tutorial, the two instances will be referred to with the following URLs:

  - [http://192.0.2.1:7200/](http://192.0.2.1:7200/) for the first instance;
  - [http://192.0.2.2:7200/](http://192.0.2.2:7200/) for the second instance.

Please adjust the URLs according to the IPs or hostnames of your own machines.

7.8.1.2 Managing repositories

Create a repository

Repositories can be created by providing a `.ttl` file with all the configuration parameters.

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

   ```bash
   curl -X POST
   http://192.0.2.1:7200/rest/repositories\n   -H 'Content-Type: multipart/form-data'\n   -F "config=repo-config.ttl"
   ```

   **Note:** You can provide a parameter `location` to create a repository in another location, see *Managing locations* below.
List repositories

Use the following cURL command to list all repositories by sending a GET request to the API:

```
curl -G http://192.0.2.1:7200/rest/repositories
   -H 'Accept: application/json'
```

The output shows the repository `repo1` that was created in the previous step.

```
[{
   "id": "repo1",
   "title": "my repository number one",
   "uri": "http://192.0.2.1:7200/repositories/repo1",
   "type": "free",
   "sesameType": "graphdb:SailRepository",
   "location": "",
   "readable": true,
   "writable": true,
   "local": true
}]
```

7.8.1.3 Managing locations

Attach a location

Use the following cURL command to attach a remote location by sending a PUT request to the API:

```
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` are optional.

List locations

Use the following cURL command to list all locations that are attached to a machine 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,
```

(continues on next page)
Note: If you skipped the “Attaching a remote location” step or if you already had other locations attached, the output will look different.

Detach a location

Use the following cURL command to detach a location from a machine by sending a DELETE request to the API:

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

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

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

7.8.1.4 Further reading

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

7.8.2 Cluster management with the GraphDB REST API

GraphDB supports a high-availability cluster based on the Raft consensus algorithm. See how to create and monitor a cluster via the GraphDB REST API here.
7.8.3 GraphDB REST API cURL commands

This page displays GraphDB REST API calls as cURL commands, which enables developers to 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.

7.8.3.1 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": 1500
} '<base_url>/rest/cluster/config'

Add cluster node

POST /rest/cluster/config/node

Example:
**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": 1500
}' <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
```

### 7.8.3.2 Data import

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 filenames that are to be imported.
- **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, interrupting.
  - `timestamp` (integer): When was the import 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.
  * normalizeDataTypeValues (boolean): Normalize recognized datatypes values.
  * normalizeLanguageTags (boolean): Normalize recognized language tags.
  * 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 recognized datatypes.
  * verifyLanguageTags (boolean): Verify language based on a given set of definitions for valid languages.

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

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

**Hint:** Common parameters:

<base_url>: The URL host and path leading to the deployed GraphDB Workbench webapp;
<repo_id>: The id string with which the current repository can be referred to;
<encoded_filepath>: Encoded filepath leading to a server file that is in the process of being imported.
7.8.3.3 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:

```bash
curl <base_url>/rest/locations
```

**Modify settings for a connected GraphDB location**

**POST /rest/locations**

Example:

```bash
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:

```bash
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>
}

7.8.3.4 Recovery management

Create, edit, and delete backups, and restore a GraphDB instance from a backup.

Get all backups

GET /rest/recovery/backups

Example:
curl '<base_url>/rest/recovery/backups'

Get backup content

GET /rest/recovery/backups/<backup_name>

Example:
curl '<base_url>/rest/recovery/backups/<backup_name>'

Create a new backup of a GraphDB instance

POST /rest/recovery/backups/<backup_name>

Example:
curl -X POST --header 'Content-Type: application/json' '<base_url>/rest/recovery/backups/<backup_name>?type=FULL&withSystemData=true'

Note: Currently, only full backup is supported.

Restore a GraphDB instance from a backup

POST /rest/recovery/restore/<backup_name>

Example:
curl -X POST --header 'Content-Type: application/json' '<base_url>/rest/recovery/restore/<backup_name>?cleanDataDir=false&type=FULL&withSystemData=true'

Note: Currently, only full backup is supported.

Update or create a backup
PUT /rest/recovery/backups/<backup_name>

Example:

curl -X PUT --header 'Content-Type: application/json' '{base_url}/rest/recovery/backups/<backup_name>?type=FULL&withSystemData=true'

**Note:** Currently, only full backup is supported.

**Delete a backup**

DELETE /rest/recovery/backups/<backup_name>

Example:

curl -X DELETE '{base_url}/rest/recovery/backups/<backup_name>'

### 7.8.3.5 Repository management

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 that the repository can be accessed at by an external service.
- **id** (string): The repository id.
- **local** (boolean): True if the repository is local (on the same machine as the Workbench).
- **location** (string): If remote, the repository’s location.
- **title** (string): The repository title.
- **type** (string): Repository type - worker, master or system.
- **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.

**Get all repositories in the current or another location**

GET /rest/repositories

Example:

curl {base_url}/rest/repositories

**Get repository configuration as Turtle**

GET /rest/repositories/<repo_id>

Example:

curl {base_url}/rest/repositories/<repo_id>?

**Get repository size**

GET /rest/repositories/<repo_id>/size

Example:
Create a repository in an attached GraphDB location (.ttl file)

POST /rest/repositories

Example:

```bash
curl -X POST <base_url>/rest/repositories?location=<encoded_location_uri> -H 'Content-Type: multipart/form-data' -F config=@<repo_ttl_config_filename>
```

Restart a repository

POST /rest/repositories/<repo_id>/restart

Example:

```bash
curl -X POST <base_url>/rest/repositories/<repo_id>/restart
```

Edit repository configuration

PUT /rest/repositories/<repo_id>

Example:

```bash
curl -X PUT <base_url>/rest/repositories/<repo_id> -H 'Accept: application/json' -H 'Content-Type: application/json' -d ' {
   "id": "<repo_id>",
   "location": "<location_uri>",
   "params": {
      "baseURL": {
         "label": "Base URL",
         "name": "baseURL",
         "value": "http://example.org/graphdb"
      },
      "entityIndexSize": {
         "label": "Entity index size",
         "name": "entityIndexSize",
         "value": "200000"
      },
      "entityIdSize": {
         "label": "Entity ID bit-size",
         "name": "entityIdSize",
         "value": "32"
      },
      "ruleset": {
         "label": "Ruleset",
         "name": "ruleset",
         "value": "owl-horst-optimized"
      },
      "storageFolder": {
         "label": "Storage folder",
         "name": "storageFolder",
         "value": "storage"
      },
      "enableContextIndex": {
         "label": "Use context index",
         "name": "enableContextIndex",
         "value": "false"
      },
      "cacheMemory": {
```
"label": "Total cache memory",
"name": "cacheMemory",
"value": "80m"
},
"tupleIndexMemory": {
"label": "Tuple index memory",
"name": "tupleIndexMemory",
"value": "80m"
},
"enablePredicateList": {
"label": "Use predicate indices",
"name": "enablePredicateList",
"value": "false"
},
"predicateMemory": {
"label": "Predicate index memory",
"name": "predicateMemory",
"value": "0"
},
"ftsMemory": {
"label": "Full-text search memory",
"name": "ftsMemory",
"value": "0"
},
"ftsIndexPolicy": {
"label": "Full-text search indexing policy",
"name": "ftsIndexPolicy",
"value": "never"
},
"ftsLiteralsOnly": {
"label": "Full-text search literals only",
"name": "ftsLiteralsOnly",
"value": "true"
},
"inMemoryLiteralProperties": {
"label": "Cache literal language tags",
"name": "inMemoryLiteralProperties",
"value": "false"
},
"enableLiteralIndex": {
"label": "Enable literal index",
"name": "enableLiteralIndex",
"value": "true"
},
"indexCompressionRatio": {
"label": "Index compression ratio",
"name": "indexCompressionRatio",
"value": "-1"
},
"checkForInconsistencies": {
"label": "Check for inconsistencies",
"name": "checkForInconsistencies",
"value": "false"
},
"disableSameAs": {
"label": "Disable owl:sameAs",
"name": "disableSameAs",
"value": "false"
},
"enableOptimization": {
"label": "Enable query optimisation",
"name": "enableOptimization",
"value": "false"
}
"name": "enableOptimization",
"value": "true"
},
"transactionIsolation": {
  "label": "Transaction isolation",
  "name": "transactionIsolation",
  "value": "true"
},
"transactionMode": {
  "label": "Transaction mode",
  "name": "transactionMode",
  "value": "safe"
},
"queryTimeout": {
  "label": "Query time-out (seconds)",
  "name": "queryTimeout",
  "value": "0"
},
"queryLimitResults": {
  "label": "Limit query results",
  "name": "queryLimitResults",
  "value": "0"
},
"throwQueryEvaluationExceptionOnTimeout": {
  "label": "Throw exception on query time-out",
  "name": "throwQueryEvaluationExceptionOnTimeout",
  "value": "false"
},
"readOnly": {
  "label": "Read-only",
  "name": "readOnly",
  "value": "false"
},
"nonInterpretablePredicates": {
  "label": "Non-interpretable predicates",
  "name": "nonInterpretablePredicates",
  "value": "http://www.w3.org/2000/01/rdf-schema#label;http://www.w3.org/1999/02/22-rdf-syntax-ns#type;http://www.ontotext.com/owlim/ces#gazetteerConfig;http://www.ontotext.com/owlim/ces#metadataConfig"
}
},
"title": "<repo_title>",
"type": "<repo_type>"
'}

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

Delete a repository in an attached GraphDB location

DELETE /rest/repositories/<repo_id>

Example:

curl -X DELETE <base_url>/rest/repositories/<repo_id>?location=<encoded_location_uri>

Hint: Common parameters:

<base_url>: The URL host and path leading to the deployed GDB Workbench web app;
7.8.3.6 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>
```
7.8.3.7 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

Example:

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' --header 'Accept: */*' -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' --header 'Accept: application/json' -d '{
    appSettings: {
        DEFAULT_SAMEAS: boolean
        DEFAULT_INFERENCE: boolean
        EXECUTE_COUNT: boolean
        IGNORE_SHARED_QUERIES: boolean
    },
    password: "<password>"
}
' <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"],
    pass: "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' --header 'Accept: text/plain' -d '{
    appSettings: {
        DEFAULT_SAMEAS: boolean
        DEFAULT_INFERENCE: boolean
        EXECUTE_COUNT: boolean
        IGNORE_SHARED_QUERIES: boolean
    },
    grantedAuthorities: ["string"],
    pass: "string",
    username: "string",
    password: "string"
}
' <base_url>/rest/security/users/<username>
```
7.8.3.8 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:

```bash
curl "<base_url>/rest/repositories/<repo_id>/sparql-templates"
```

**Get a SPARQL template configuration**

```
GET /rest/repositories/<repo_id>/sparql-templates/configuration
```

Example:

```bash
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:

```bash
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:

```bash
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:

```bash
curl -X PUT --header 'Content-Type: text/plain' --header 'Accept: */*' -d '"update_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):

```bash
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
"sugar" : "none",
"year" : 2020,
"s" : "http://www.ontotext.com/example/wine#Blanquito"
} ' '<base_url>/rest/repositories/<repo_id>/sparql-templates/execute'
```
7.8.3.9 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>' --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>
```
7.8.3.10 Authentication

Obtain a GDB token in exchange for username and password

```http
POST /rest/login/**
```

Example:

```bash
curl -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.

7.9 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.

7.9.1 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
<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;"/>

<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 = ` 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 {
  ?related_entity sesame:directType ?type .
}
```

(continues on next page)
?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 )
  {
    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 .
    FILTER(?p2 NOT IN (dbo:wikiPageWikiLink)) .
    ?related_entity sesame:directType ?entity_type .
    ?related_entity rank:hasRDFRank ?related_entity_rank .
  }
  UNION
  {
    ?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) {
    });
    var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
      return t[0]
    })), _.map(linkTriples, function (t) { (continues on next page)
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]) + ")" : "") + getLocalName(type[2]) : ":" + getLocalName(type[2]) : "#ecceef")},
    color: ((type != undefined) ? stringToColour(type[2]) : ":#ecceef")
    }
});

// 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', {}, { (continues on next page)
// sync parameters
onEnd: endLayout

function endLayout() {
  ogma.locate.center({
    easing: 'linear',
    duration: 300
  });
  endLayout();
}

Which produces the following graph:
7.9.2 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
<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;">

<script>
// Which namespace to chose types from
var dboNamespace = "http://dbpedia.org/ontology"

var suspiciousOffshore = '# F05: Suspicious control chain through off-shore company

PREFIX onto: <http://www.ontotext.com/>
PREFIX ff-map: <http://factforge.net/ff2016-mapping/>
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
PREFIX dbo: <http://dbpedia.org/ontology/>

CONSTRUCT {
    ?c1 ff-map:primaryCountry ?c1_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 sesame:directType ?t1 .
    ?c2 sesame:directType ?t2 .
    ?c3 sesame:directType ?t3 .
    ?c1 ff-map:primaryCountry ?c1_country .
    ?c3 ff-map:primaryCountry ?c1_country .
    FILTER (?c1_country != ?c2_country)
    }
}

var postData = {
(continues on next page)
```

7.9. Visualize GraphDB Data with Ogma JS
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
  });
  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
```javascript
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() {
  ogma.locate.center({
    easing: 'linear',
    duration: 300
  });
}
</script>
</body>
</html>

Which produces the following graph:
7.9.3 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.
7.9. Visualize GraphDB Data with Ogma JS
success: function (data) {

    var triples = convertData(data);

    // Get all nodes uris
    var linkTriples = _.filter(triples, function (triple) {
    });
    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),
            size: 0.5,
            color: ((type !== undefined) ? stringToColour(type[2]) : "#ceceef"),
            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]),
        }
    });
}

(continues on next page)
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
      sizeZoomReferential: 5, // Paris will be displayed with a radius
      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) {
      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;
})

(continues on next page)
Which produces the following graph:

![Graph Image](image)

### 7.9.4 Common function to visualize GraphDB data

The `commons.js` file used by all demos:

```javascript
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);
    }
}
```

(continues on next page)
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.

7.10 Create 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 relationships 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.
7.10.1 How it works?

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.

7.10.2 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.

7.10.2.1 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.
7.10.2.2 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:

```sparql
PREFIX onto: <http://www.ontotext.com/>
construct {
    ?source onto:hasFlightTo ?destination .
} where {
    ?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.

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:

```sparql
PREFIX onto: <http://www.ontotext.com/>
construct {
    ?node onto:hasFlightTo ?destination .
} where {
} 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 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:

```sparql
PREFIX onto: <http://www.ontotext.com/>
construct {
    ?airport onto:hasFlightFromWithAirline ?airline .
} where {
} limit 100
```

And let’s create a visual graph with the following expand query:

```sparql
# 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 {

(continues on next page)
# 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

?a1 onto:hasFlightFromWithAirline ?node .
?a2 onto:hasFlightToWithAirline ?node .

WHERE {

} UNION {

} UNION {

} UNION {

} UNION {

7.10.3 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.

7.10.3.1 Data model

This is the full data model:

![Data model diagram]

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.

7.10. Create Custom Graph View over Your RDF Data
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.

```sparql
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.
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.

```sql
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(?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:

```sql
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), "," , "")) as ?researcherName)
  optional {?node ?p ?o}
  ?contribution sg:hasAffiliation ?affiliation .
  ?affiliation sg:publishedName ?value .
} limit 100
```
7.10.4 Additional sources

To learn more about the SPARQL editing and data visualization capabilities of the GraphDB Workbench, as well as features that can be added with little programming, and about SPARQL writing aids and visualization tools that can be integrated with GraphDB, please have a look at this How-to Guide.

7.11 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.

7.11.1 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

<table>
<thead>
<tr>
<th>Description</th>
<th>onto:implicit</th>
<th>The graph contains statements inferred via the repository’s ruleset, located in the default graph.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>onto:explicit</td>
<td>The graph contains statements inserted in the database by the user, located in the default graph.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:readonly</td>
<td>The graph contains schematic statements, i.e., the statements which define the repository’s ruleset.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:readwrite</td>
<td>The graph contains non-schematic statements - i.e., all statements beside the ones in the ruleset.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:count</td>
<td>A pseudo graph that forces a count of the results of the query and returns it as the result.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:disable-sameAs</td>
<td>A pseudo graph that disables the default behavior of expanding the sameAs nodes of the query result.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:distinct</td>
<td>A pseudo graph that makes the query behave as if the DISTINCT keyword was used.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:skip-redundant-implicit</td>
<td>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.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:merge</td>
<td>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.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:hash</td>
<td>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.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:explain</td>
<td>A pseudo graph that returns the query optimization plan.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:commitStatistics</td>
<td>If enabled, logs commit statistics every 30 seconds.</td>
</tr>
<tr>
<td>Description</td>
<td>sys:statistics</td>
<td>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.</td>
</tr>
<tr>
<td>Description</td>
<td>onto:retain-bind-position</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 7.11.2 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

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sys:schemaTransaction</code></td>
<td>Allows for axiom insertion and removal, changing the ruleset.</td>
</tr>
<tr>
<td><code>sys:reinfer</code></td>
<td>Forces full inference re-computation.</td>
</tr>
<tr>
<td><code>sys:turnInferenceOn</code></td>
<td>Enables inferences.</td>
</tr>
<tr>
<td><code>sys:turnInferenceOff</code></td>
<td>Disables inference. This will not remove previously inferred statements.</td>
</tr>
<tr>
<td><code>sys:addRuleSet</code></td>
<td>Adds a ruleset.</td>
</tr>
<tr>
<td><code>sys:removeRuleSet</code></td>
<td>Removes a ruleset.</td>
</tr>
<tr>
<td><code>sys:defaultRuleSet</code></td>
<td>Refers to the default ruleset. Can be used to fetch it or change it.</td>
</tr>
<tr>
<td><code>sys:currentRuleSet</code></td>
<td>Refers to the current ruleset. Can be used to fetch it or change it.</td>
</tr>
<tr>
<td><code>sys:listRuleSets</code></td>
<td>Lists the currently installed rulesets.</td>
</tr>
<tr>
<td><code>sys:renameRuleSet</code></td>
<td>Renames a ruleset.</td>
</tr>
<tr>
<td><code>sys:exploreRuleSet</code></td>
<td>Retrieves a ruleset’s text, if any.</td>
</tr>
<tr>
<td><code>sys:consistencyCheckAgainstRuleSet</code></td>
<td>Checks data consistency against a given ruleset.</td>
</tr>
<tr>
<td><code>onto:replaceGraph</code></td>
<td>Sets a replacement graph. The content of that graph will be replaced with</td>
</tr>
<tr>
<td></td>
<td>the incoming update. Multiple graphs may be provided by multiple calls</td>
</tr>
<tr>
<td></td>
<td>with this predicate.</td>
</tr>
<tr>
<td><code>onto:replaceGraphPrefix</code></td>
<td>Sets a prefix for replacement graphs. All graphs whose IRIs start with the</td>
</tr>
<tr>
<td></td>
<td>prefix will be replaced with the incoming update. Multiple prefixes may</td>
</tr>
<tr>
<td></td>
<td>be provided by multiple calls with this predicate.</td>
</tr>
</tbody>
</table>

### 7.12 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.

#### 7.12.1 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: `-P0Y0M25DT4H29M58.999S`. 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 `-P0Y0M25DT4H29M58.999S` is passed to them:
### Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Expected return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofn:years-from-duration</td>
<td>Return the “years” part of the duration literal</td>
<td>0</td>
</tr>
<tr>
<td>ofn:months-from-duration</td>
<td>Return the “months” part of the duration literal</td>
<td>0</td>
</tr>
<tr>
<td>ofn:days-from-duration</td>
<td>Return the “days” part of the duration literal</td>
<td>25</td>
</tr>
<tr>
<td>ofn:hours-from-duration</td>
<td>Return the “hours” part of the duration literal</td>
<td>4</td>
</tr>
<tr>
<td>ofn:minutes-from-duration</td>
<td>Return the “minutes” part of the duration literal</td>
<td>29</td>
</tr>
<tr>
<td>ofn:seconds-from-duration</td>
<td>Return the “seconds” part of the duration literal</td>
<td>58</td>
</tr>
<tr>
<td>ofn:millis-from-duration</td>
<td>Returns the “milliseconds” part of the duration literal</td>
<td>999</td>
</tr>
</tbody>
</table>

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("-P0Y0M25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```

### 7.12.2 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, `-P0Y0M25DT4H29M58.999S` 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 `-P0Y0M25DT4H29M58.999S` is passed to them. Note that the return values are negative since the period is negative:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Expected return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofn:asWeeks</td>
<td>Returns the duration of the period as weeks</td>
<td>-3</td>
</tr>
<tr>
<td>ofn:asDays</td>
<td>Returns the duration of the period as days</td>
<td>-25</td>
</tr>
<tr>
<td>ofn:asHours</td>
<td>Returns the duration of the period as hours</td>
<td>-604</td>
</tr>
<tr>
<td>ofn:asMinutes</td>
<td>Returns the duration of the period as minutes</td>
<td>-36269</td>
</tr>
<tr>
<td>ofn:asSeconds</td>
<td>Returns the duration of the period as seconds</td>
<td>-2176198</td>
</tr>
<tr>
<td>ofn:asMillis</td>
<td>Returns the duration of the period as milliseconds</td>
<td>-2176198999</td>
</tr>
</tbody>
</table>

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("-P0Y0M25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```
7.12.3 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Expected return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofn:weeksBetween</td>
<td>Returns the duration between the two dates as weeks</td>
<td>3</td>
</tr>
<tr>
<td>ofn:daysBetween</td>
<td>Returns the duration between the two dates as days</td>
<td>25</td>
</tr>
<tr>
<td>ofn:hoursBetween</td>
<td>Returns the duration between the two dates as hours</td>
<td>604</td>
</tr>
<tr>
<td>ofn:minutesBetween</td>
<td>Returns the duration between the two dates as minutes</td>
<td>36269</td>
</tr>
<tr>
<td>ofn:secondsBetween</td>
<td>Returns the duration between the two dates as seconds</td>
<td>2176198</td>
</tr>
<tr>
<td>ofn:millisBetween</td>
<td>Returns the duration between the two dates as milliseconds</td>
<td>2176198999</td>
</tr>
</tbody>
</table>

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 {
}
```

7.12.4 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.

7.13 GraphDB Fundamentals

GraphDB Fundamentals builds the bases 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.
7.13.1 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.

7.13.2 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 your first RDF graph and run your first SPARQL queries.

7.13.3 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.

7.13.4 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.

7.13.5 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.

7.13.6 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.

7.13.7 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.
7.13.8 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.

7.13.9 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.

7.13.10 Module 10: Connectors

The Lucene, Solr, and Elasticsearch 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.

The GraphDB Developer Hub is meant as the central point for the GraphDB Developer Community. It serves as a hands-on compendium to the GraphDB Documentation that gives practical advice and tips on accomplishing real-world tasks.

If you are new to RDF databases, you should start with the basics:

- Data Modeling with RDF(S)
- SPARQL
- RDF-star and SPARQL-star
- Ontologies
- Inference

If you are already familiar with RDF or are eager to start programming, please refer to:

- Programming with GraphDB
- Extending GraphDB Workbench
- GraphDB REST API
- Visualize GraphDB Data with Ogma JS
- Create Custom Graph View over Your RDF Data

If you want documentation on GraphDB extension functions:

- GraphDB System Statements
- Time Functions Extensions

If you want an in-depth introduction to everything GraphDB, we suggest the following video tutorials:

- GraphDB Fundamentals
8.1 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.

8.1.1 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.

**8.1.1.1 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](http://www.johndoesite.com/aboutme.htm) 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.

**8.1.1.2 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:


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:

\[
\text{<jds: dc:creator jds:aboutme>}
\]

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 \( \text{foo} \) 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 \( \text{jds} \) for http://www.johndoesite.com and use the Dublin Core Metadata namespace \( \text{dc} \) for http://purl.org/dc/elements/1.1/.

So, the shorthand form for the example statement is simply:

\[
\text{jds: dc:creator jds:aboutme}
\]

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
<?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: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.
8.1.3 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.

8.1.4 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.

8.1.2 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.
8.1.2.1 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.

8.1.2.2 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 externs:weightInKg, is defined with the following RDF statement:

```
externs: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.
8.1.2.3 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.

8.1.2.4 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:description>Describes planting and nurturing rose bushes.</dc:description>
  </rdf:Description>
</rdf:RDF>
```
8.1.3 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.

\[
O = \langle C, R, I, A \rangle
\]

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).

### 8.1.3.1 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.

### 8.1.3.2 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.

8.1.4 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.

8.1.4.1 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.

8.1.4.2 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:

\[ A_1, \ldots, A_n \rightarrow B \]

where \( A_i \) and \( B \) are atomic formulas. In fact, there are two intuitive ways of reading such a rule:

• If \( A_1, \ldots, A_n \) are known to be true, then \( B \) is also true. Rules with this interpretation are referred to as ‘deductive rules’.
• If the conditions \( A_1, \ldots, A_n \) 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.
8.1.4.3 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.

8.1.5 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

8.1.5.1 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

Expressivity Supported in GraphDB

- OWL Full
- SWRL
- OWL DL
- OWL Lite
- Datalog
- OWL 2 RL
- OWL 2 QL
- OWL Horst
- RDFS

Complexity*

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.

### 8.1.5.2 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.

### 8.1.5.3 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 expressivity. 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.
8.1.5.4 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.

8.1.5.5 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.

8.1.6 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.

8.1.6.1 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).
8.1.6.2 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.

8.1.6.3 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.

8.1.7 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.
8.1.7.1 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).

8.1.8 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.
# 8.2 GraphDB Feature Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>GraphDB Free</th>
<th>GraphDB SE</th>
<th>GraphDB EE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage unlimited number of RDF statements</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Full SPARQL 1.1 support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deploy anywhere using Java</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>100% compatible with RDF4J framework</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ultra fast forward-chaining reasoning</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Efficient retraction of inferred statements upon update</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Full standard-compliant and optimized rulesets for RDFS, OWL 2 RL, and QL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Custom reasoning and consistency checking rulesets</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Plugin API for engine extension</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Support for Geospatial indexing &amp; querying, plus GeoSPARQL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Query optimizer allowing effective query execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Workbench interface to manage repositories, data, user accounts and access roles</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lucene connector for full-text search</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Solr connector for full-text search</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Elasticsearch connector for full-text search</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Kafka connector for synchronizing changes to the RDF model to any Kafka consumer</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>High performance load, query and inference simultaneously</td>
<td>Limited to two concurrent queries</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Automatic failover, synchronization and load balancing to maximize cluster utilisation</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Scale out concurrent query processing allowing query throughput to scale proportionally to the number of cluster nodes</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Cluster elasticity remaining fully functional in the event of failing nodes</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Community support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Commercial SLA</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

# 8.3 Repository Configuration Template - How It 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.

**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 namespaces corresponding to the prefixes used in the above paragraph are as follows:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rep:</code></td>
<td><a href="http://www.openrdf.org/config/repository#">http://www.openrdf.org/config/repository#</a></td>
</tr>
<tr>
<td><code>sr:</code></td>
<td><a href="http://www.openrdf.org/config/repository/sail#">http://www.openrdf.org/config/repository/sail#</a></td>
</tr>
<tr>
<td><code>sail:</code></td>
<td><a href="http://www.openrdf.org/config/sail#">http://www.openrdf.org/config/sail#</a></td>
</tr>
<tr>
<td><code>graphdb:</code></td>
<td><a href="http://www.ontotext.com/trree/graphdb#">http://www.ontotext.com/trree/graphdb#</a></td>
</tr>
</tbody>
</table>

All properties used to specify the GraphDB configuration parameters use the `graphdb:` prefix and the local names match up with the configuration parameters, e.g., the value of the `ruleset` parameter can be specified using the `graphdb:ruleset` property.
8.4 Ontology Mapping with \texttt{owl:sameAs} Property

GraphDB’s \texttt{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:

\begin{verbatim}
SELECT * {
    UNION
    UNION
    {<http://sws.geonames.org/727011/> ?p ?o .}
    UNION
    {<http://rdf.freebase.com/ns/m/0ftjx> ?p ?o .}
}
\end{verbatim}

Or you can even use a shorter one:

\begin{verbatim}
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>))
}
\end{verbatim}

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., \url{http://dbpedia.org/resource/Plovdiv}, \url{http://sws.geonames.org/653987/}, and \url{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:

\begin{verbatim}
SELECT * {
    ?s ?p1 ?x .
    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://dbpedia.org/property/instance_of>,
            <http://dbpedia.org/property/label>)
}
\end{verbatim}

(continues on next page)
But if you can say through rules and assertions that given URIs are the same, then you can simply write:

```
SELECT * {
}
```

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:

```
```

then you can conclude with the given rules that:

```
```

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.
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 the 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 [here](#).

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 [here](#).

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. There are two ways to do this:

- By editing the repository configuration
- By importing the repository configuration file in the Workbench (does not require stopping GraphDB)

See more about it [here](#).

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.

8.5.5.1 URL patterns for this new functionality are provided at

- `<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 supported RDF formats.
- **Content-Type**: Must specify the encoding of any request data sent to a server. Relevant values are the MIME types of 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.

8.6 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:boolean BOUND(variable var)</code></td>
<td>Returns <code>true</code> if the variable <code>var</code> is bound to a value. Returns <code>false</code> otherwise. Variables with the value NaN or INF are considered bound. More</td>
</tr>
<tr>
<td><code>rdfTerm IF(expression e1, expression e2, expression e3)</code></td>
<td>The IF function form evaluates the first argument, interprets it as a effective boolean value, then returns the value of <code>e2</code> if the EBV is true, otherwise it returns the value of <code>e3</code>. Only one of <code>e2</code> and <code>e3</code> is evaluated. If evaluating the first argument raises an error, then an error is raised for the evaluation of the IF expression. More</td>
</tr>
<tr>
<td><code>rdfTerm COALESCE(expression e1, ...)</code></td>
<td>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</td>
</tr>
<tr>
<td><code>xsd:boolean NOT EXISTS { pattern }</code></td>
<td>There is a filter operator EXISTS that takes a graph pattern. EXISTS returns <code>true</code> or <code>false</code> 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 <code>fn:not(EXISTS{...})</code>. More</td>
</tr>
<tr>
<td>`xsd:boolean xsd:boolean left</td>
<td></td>
</tr>
<tr>
<td><code>xsd:boolean xsd:boolean left &amp; xsd:boolean right</code></td>
<td>Returns a logical AND of <code>left</code> and <code>right</code>. Note that logical-and operates on the effective boolean value of its arguments. More</td>
</tr>
<tr>
<td><code>xsd:boolean rdfTerm term1 = rdfTerm term2</code></td>
<td>Returns <code>true</code> if <code>term1</code> and <code>term2</code> are equal. Returns <code>false</code> 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</td>
</tr>
<tr>
<td><code>xsd:boolean sameTerm(rdfTerm term1, rdfTerm term2)</code></td>
<td>Returns <code>true</code> if <code>term1</code> and <code>term2</code> are the same RDF term as defined in RDF Concepts; returns <code>false</code> otherwise. More</td>
</tr>
<tr>
<td><code>xsd:boolean rdfTerm term IN (expression e1, ...)</code></td>
<td>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 <code>=</code> operator, which compares the RDF term to each expression for equality. More</td>
</tr>
<tr>
<td><code>xsd:boolean NOT IN (expression e1, ...)</code></td>
<td>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 <code>!=</code> operator, which compares the RDF term to each expression for inequality. More</td>
</tr>
<tr>
<td><code>xsd:boolean isIRI(rdfTerm term)</code></td>
<td>Returns <code>true</code> if <code>term</code> is an IRI. Returns <code>false</code> otherwise. More</td>
</tr>
<tr>
<td><code>xsd:boolean isURI(rdfTerm term)</code></td>
<td>Returns <code>true</code> if <code>term</code> is an URI. Returns <code>false</code> otherwise. More</td>
</tr>
<tr>
<td><code>xsd:boolean isBlank(rdfTerm term)</code></td>
<td>Returns <code>true</code> if <code>term</code> is a blank node. Returns <code>false</code> otherwise. More</td>
</tr>
<tr>
<td><code>xsd:boolean isLiteral(rdfTerm term)</code></td>
<td>Returns <code>true</code> if <code>term</code> is a literal. Returns <code>false</code> otherwise. More</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:boolean isNumeric(rdfTerm term)</td>
<td>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)</td>
</tr>
<tr>
<td>xsd:string STR(literal ltrl)</td>
<td>Returns the lexical form of ltrl (a literal); returns the codepoint representation of rsoc (an IRI). This is useful for examining parts of an IRI, for instance, the hostname. (More)</td>
</tr>
<tr>
<td>xsd:string STR(iri rsoc)</td>
<td></td>
</tr>
<tr>
<td>xsd:string LANG(literal ltrl)</td>
<td>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)</td>
</tr>
<tr>
<td>iri DATATYPE(literal ltrl)</td>
<td>Returns the datatype IRI of the literal ltrl. (More)</td>
</tr>
<tr>
<td>iri IRI(string str)</td>
<td>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 rsoc, it returns the IRI unchanged. (More)</td>
</tr>
<tr>
<td>iri IRI(iri rsoc)</td>
<td></td>
</tr>
<tr>
<td>iri URI(string str)</td>
<td></td>
</tr>
<tr>
<td>iri URI(iri rsoc)</td>
<td></td>
</tr>
<tr>
<td>bnode BNODE()</td>
<td>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)</td>
</tr>
<tr>
<td>bnode BNODE(string str)</td>
<td></td>
</tr>
<tr>
<td>literal STRDT(xsd:string lexicalForm, iri datatypeIRI)</td>
<td>The STRDT function constructs a literal with lexical form and type as specified by the arguments. (More)</td>
</tr>
<tr>
<td>xsd:langString STRLANG(xsd:string lexicalForm, xsd:string langTag)</td>
<td>The STRLANG function constructs a literal with lexical form and language tag as specified by the arguments. (More)</td>
</tr>
<tr>
<td>iri UUID()</td>
<td>Return a fresh IRI from the UUID URN scheme. Each call of UUID() returns a different UUID. (More)</td>
</tr>
<tr>
<td>xsd:string STRUUID()</td>
<td>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)</td>
</tr>
<tr>
<td>xsd:integer STRLEN(string str)</td>
<td>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)</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string SUBSTR(string source, xsd:integer startingLoc)</td>
<td>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</td>
</tr>
<tr>
<td>string SUBSTR(string source, xsd:integer startingLoc, xsd:integer length)</td>
<td></td>
</tr>
<tr>
<td>string UCASE(string str)</td>
<td>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</td>
</tr>
<tr>
<td>string LCASE(string str)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:boolean STRSTARTS(string str1, string str2)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:boolean STRENDS(string str1, string str2)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:boolean CONTAINS(string str1, string str2)</td>
<td>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</td>
</tr>
<tr>
<td>string STRBEFORE(string str1, string str2)</td>
<td>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</td>
</tr>
<tr>
<td>string STRAFTER(string str1, string str2)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:string ENCODE_FOR_URI(string str)</td>
<td>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</td>
</tr>
<tr>
<td>string CONCAT(string str1, ...)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:boolean lang-Matches(xsd:string languageTag, xsd:string languageRange)</td>
<td>Returns true if languageTag (first argument) matches languageRange (second argument). languageRange is a basic language range. A language range of &quot;*&quot; matches any non-empty language tag string. More</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>xsd:boolean REGEX(string text, xsd:string pattern)</td>
<td>Invokes the XPath fn:matches function to match text against a regular expression pattern. More</td>
</tr>
<tr>
<td>xsd:boolean REGEX(string text, xsd:string pattern, xsd:string flags)</td>
<td></td>
</tr>
<tr>
<td>string REPLACE(string arg, xsd:string pattern, xsd:string replacement)</td>
<td>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 modifier flags. More</td>
</tr>
<tr>
<td>string REPLACE(string arg, xsd:string pattern, xsd:string replacement, xsd:string flags)</td>
<td></td>
</tr>
<tr>
<td>numeric ABS(numeric num)</td>
<td>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 XDM. More</td>
</tr>
<tr>
<td>numeric ROUND(numeric num)</td>
<td>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 XDM. More</td>
</tr>
<tr>
<td>numeric CEIL(numeric num)</td>
<td>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 XDM. More</td>
</tr>
<tr>
<td>numeric FLOOR(numeric num)</td>
<td>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 XDM. More</td>
</tr>
<tr>
<td>xsd:double RAND()</td>
<td>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</td>
</tr>
<tr>
<td>xsd:dateTime NOW()</td>
<td>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</td>
</tr>
<tr>
<td>xsd:integer YEAR(xsd:dateTime arg)</td>
<td>Returns the year part of arg as an integer. This function corresponds to fn:year-from-dateTime. More</td>
</tr>
<tr>
<td>xsd:integer MONTH(xsd:dateTime arg)</td>
<td>Returns the month part of arg as an integer. This function corresponds to fn:month-from-dateTime. More</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:integer</td>
<td>Returns the day part of ( arg ) as an integer. This function corresponds to ( \text{fn:day-from-dateTime} ). <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{DAY(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:integer</td>
<td>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 ( \text{fn:hours-from-dateTime} ). <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{HOURS(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:integer</td>
<td>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 ( \text{fn:minutes-from-dateTime} ). <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{MINUTES(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:decimal</td>
<td>Returns the seconds part of the lexical form of ( arg ). This function corresponds to ( \text{fn:seconds-from-dateTime} ). <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{SECONDS(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:dayTimeDuration</td>
<td>Returns the timezone part of ( arg ) as an xsd:dayTimeDuration. Raises an error if there is no timezone. This function corresponds to ( \text{fn:timezone-from-dateTime} ) except for the treatment of literals with no timezone. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{TIMEZONE(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:string</td>
<td>Returns the timezone part of ( arg ) as a simple literal. Returns the empty string if there is no timezone. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{TZ(xsd:dateTime arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:string</td>
<td>Returns the MD5 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{MD5(xsd:string arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:string</td>
<td>Returns the SHA1 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{SHA1(xsd:string arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:string</td>
<td>Returns the SHA256 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{SHA256(xsd:string arg)} )</td>
<td></td>
</tr>
<tr>
<td>xsd:string</td>
<td>Returns the SHA512 checksum, as a hex digit string, calculated on the UTF-8 representation of the lexical form of the argument. <a href="#">More</a></td>
</tr>
<tr>
<td>( \text{SHA512(xsd:string arg)} )</td>
<td></td>
</tr>
</tbody>
</table>

### 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 \( \text{xsd:date} \) constructor.
### Constructor function

<table>
<thead>
<tr>
<th>xsd:integer(rdfTerm value)</th>
<th>Casts value to xsd:integer. More</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:decimal(rdfTerm value)</td>
<td>Casts value to xsd:decimal. More</td>
</tr>
<tr>
<td>xsd:float(rdfTerm value)</td>
<td>Casts value to xsd:float. More</td>
</tr>
<tr>
<td>xsd:double(rdfTerm value)</td>
<td>Casts value to xsd:double. More</td>
</tr>
<tr>
<td>xsd:string(rdfTerm value)</td>
<td>Casts value to xsd:string. More</td>
</tr>
<tr>
<td>xsd:boolean(rdfTerm value)</td>
<td>Casts value to xsd:boolean. More</td>
</tr>
<tr>
<td>xsd:dateTime(rdfTerm value)</td>
<td>Casts value to xsd:dateTime. More</td>
</tr>
</tbody>
</table>

### 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.


<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofn:acos(numeric a)</td>
<td>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</td>
</tr>
<tr>
<td>ofn:asin(numeric a)</td>
<td>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</td>
</tr>
<tr>
<td>ofn:atan(numeric a)</td>
<td>The arctangent function. The output is in the range $(-\pi/2, \pi/2)$ radians. See Math.atan(double). Example: ofn:atan(1) = 0.7853981633974483</td>
</tr>
<tr>
<td>ofn:atan2(numeric y, numeric x)</td>
<td>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</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:double ofn:copySign(numeric magnitude, numeric sign)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:cos(numeric a)</td>
<td>The cosine function. The argument is in radians. See Math.cos(double). Example: ofn:cos(1) = 0.5403023058681398</td>
</tr>
<tr>
<td>xsd:double ofn:e()</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:expm1(numeric x)</td>
<td>Returns e^x - 1. See Math.expm1(double). Example: ofn:expm1(3) = 19.085536923187668</td>
</tr>
<tr>
<td>xsd:double ofn:floorDiv(numeric x, numeric y)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:floorMod(numeric x, numeric y)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:getExponent(numeric d)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:hypot(numeric x, numeric y)</td>
<td>Returns sqrt(x^2 + y^2) without intermediate overflow or underflow. See Math.hypot(double,double). Example: ofn:hypot(3, 4) = 5.0</td>
</tr>
<tr>
<td>xsd:double ofn:IEEEremainder(numeric f1, numeric f2)</td>
<td>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</td>
</tr>
<tr>
<td>xsd:double ofn:log10(numeric a)</td>
<td>The common (decimal) logarithm function. See Math.log10(double). Example: ofn:log10(4) = 0.6020599913279624</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 \(\pi\), 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 \(d \times 2^{\text{scaleFactor}}\) 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).  
  Example: ofn:scalb(3, 3) = 24.0 |
| 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 |
  Example: ofn:sinh(2) = 3.62686840747019 |
  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) = 1.55740772466549023 |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
  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 |                                |

### 8.6.4 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*. 

---

**Table 2 – continued from previous page**

---

8.6. SPARQL Functions Reference  

---

GraphDB Documentation, Release 10.0.0
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:long ofn:years-from-duration(xsd:duration dur)</td>
<td>Return the “years” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:months-from-duration(xsd:duration dur)</td>
<td>Returns the “months” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:days-from-duration(xsd:duration dur)</td>
<td>Returns the “days” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:hours-from-duration(xsd:duration dur)</td>
<td>Returns the “hours” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:minutes-from-duration(xsd:duration dur)</td>
<td>Returns the “minutes” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:seconds-from-duration(xsd:duration dur)</td>
<td>Returns the “seconds” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:millis-from-duration(xsd:duration dur)</td>
<td>Returns the “milliseconds” part of the duration literal</td>
</tr>
<tr>
<td>xsd:long ofn:asWeeks(xsd:duration dur)</td>
<td>Returns the duration of the period as weeks</td>
</tr>
<tr>
<td>xsd:long ofn:asDays(xsd:duration dur)</td>
<td>Returns the duration of the period as days</td>
</tr>
<tr>
<td>xsd:long ofn:asHours(xsd:duration dur)</td>
<td>Returns the duration of the period as hours</td>
</tr>
<tr>
<td>xsd:long ofn:asMinutes(xsd:duration dur)</td>
<td>Returns the duration of the period as minutes</td>
</tr>
<tr>
<td>xsd:long ofn:asSeconds(xsd:duration dur)</td>
<td>Returns the duration of the period as seconds</td>
</tr>
<tr>
<td>xsd:long ofn:asMillis(xsd:duration dur)</td>
<td>Returns the duration of the period as milliseconds</td>
</tr>
<tr>
<td>xsd:long ofn:weeksBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as weeks</td>
</tr>
<tr>
<td>xsd:long ofn:daysBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as days</td>
</tr>
<tr>
<td>xsd:long ofn:hoursBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as hours</td>
</tr>
<tr>
<td>xsd:long ofn:minutesBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as minutes</td>
</tr>
<tr>
<td>xsd:long ofn:secondsBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as seconds</td>
</tr>
<tr>
<td>xsd:long ofn:millisBetween(xsd:dateTime d1, xsd:dateTime d2)</td>
<td>Returns the duration between the two dates as milliseconds</td>
</tr>
</tbody>
</table>
8.6.5 SPARQL SPIN functions and magic predicates

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.
# 8.6.5.1 SPARQL functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dates</strong></td>
<td></td>
</tr>
<tr>
<td>spif:parseDate(xsd:string date, xsd:string format)</td>
<td>Parses date using format with Java’s SimpleDateFormat</td>
</tr>
<tr>
<td>spif:dateFormat(xsd:dateTime date, xsd:string format)</td>
<td>Formats date using format with Java SimpleDateFormat</td>
</tr>
<tr>
<td>xsd:long spif:currentTimeMillis()</td>
<td>The current time in milliseconds since the epoch</td>
</tr>
<tr>
<td>xsd:long spif:timeMillis(xsd:dateTime date)</td>
<td>The time in milliseconds since the epoch for the provided argument</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Numbers</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric spif:mod(xsd:numeric dividend, xsd:numeric divisor)</td>
<td>Remainder from integer division</td>
</tr>
<tr>
<td>xsd:string spif:decimalFormat(numeric number, xsd:string format)</td>
<td>Formats number using format with Java’s DecimalFormat</td>
</tr>
<tr>
<td>xsd:double spif:random()</td>
<td>Calls Java’s Math.random()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Strings</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:string spif:trim(string str)</td>
<td>Calls String.trim()</td>
</tr>
<tr>
<td>spif:generateUUID()</td>
<td>UUID generation as a literal. Same as SPARQL’s STRUUID().</td>
</tr>
<tr>
<td>spif:cast(literal value, iri type)</td>
<td>Same as SPARQL’s STRDT(STR(value), type). Does not do validation either.</td>
</tr>
<tr>
<td>xsd:int spif:indexOf(string str, string substr)</td>
<td>Position of first occurrence of a substring.</td>
</tr>
<tr>
<td>Note that SPIN functions that work on text use 0-based indexes, unlike SPARQL’s functions, which use 1-based indexes.</td>
<td></td>
</tr>
<tr>
<td>xsd:int spif:lastIndexOf(string str, string substr)</td>
<td>Position of last occurrence of a substring.</td>
</tr>
<tr>
<td>Note that SPIN functions that work on text use 0-based indexes, unlike SPARQL’s functions, which use 1-based indexes.</td>
<td></td>
</tr>
<tr>
<td>xsd:string spif:buildString(string template, arguments...)</td>
<td>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.</td>
</tr>
<tr>
<td>xsd:string spif:replaceAll(string str, xsd:string regexp, xsd:string flags)</td>
<td>Calls Java String.replaceAll, same as SPARQL’s REPLACE().</td>
</tr>
<tr>
<td>spif:unCamelCase(string str)</td>
<td>Converts camel-cased string to non-camel case string with spaces</td>
</tr>
<tr>
<td>spif:upperCase(string str)</td>
<td>Converts to upper case, similar to SPARQL’s UCASE() but disregarding the language tag from the input string</td>
</tr>
<tr>
<td>spif:lowerCase(string str)</td>
<td>Converts to lower case, similar to SPARQL’s LCASE() but disregarding the language tag from the input string</td>
</tr>
<tr>
<td>spif:titleCase(string str)</td>
<td>Converts to title case (each word starts with a capital). You can use this function to convert a multi-word string to CamelCase as follows: replace(titleCase(?x),&quot;\n&quot;,&quot;&quot;)</td>
</tr>
<tr>
<td>spif:lowerTitleCase(string str)</td>
<td>Converts to lower title case (each but the first word starts with a capital. You can use this function to convert a multi-word string to camelCase as follows: replace(lowerTitleCase(?x),&quot;\n&quot;,&quot;&quot;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>IRIs</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:string spif:encodeURL(string str)</td>
<td>Calls java.net.URLEncoder.encode, similar to SPARQL’s ENCODE_FOR_URI() but not the same</td>
</tr>
<tr>
<td>spif:decodeURL(string str)</td>
<td>Calls java.net.URLEncoder.decode</td>
</tr>
<tr>
<td>iri spif:buildURI(string template, arguments...)</td>
<td>Builds a URI from a template, e.g. “<a href="">urn:{?1}/{?2}</a>” where {?2} will be replaced with second argument and {?1} will be replaced with the first argument after the template. The template must begin with <code>&lt;</code> and end with <code>&gt;</code></td>
</tr>
<tr>
<td>xsd:boolean spif:isValidURI(string str)</td>
<td>According to some standards, note that it allows spaces too</td>
</tr>
</tbody>
</table>

**8.6. SPARQL Functions Reference**
8.6.5.2 Magic predicates

Magic predicates are similar to how GraphDB plugins can interpret certain triple patterns. Unlike a function, they can return multiple values per call.

The magic predicates are used as triple patterns like this: `?result :magicPredicate ("arg1" "arg2" `_)

There are three magic predicates: `spif:split`, `spif:for`, and `spif:foreach`.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?result spif:split (?string ?regex)</code></td>
<td>Takes two arguments: a string to split and a regex to split on. The current implementation uses Java's <code>String.split()</code>.</td>
</tr>
<tr>
<td><code>?result spif:for (?start ?end)</code></td>
<td>Generates bindings from a given <code>start</code> integer value to another given <code>end</code> integer value.</td>
</tr>
<tr>
<td><code>?result spif:foreach (?arg1 ?arg2 _)</code></td>
<td>Generates bindings for the given arguments <code>arg1</code>, <code>arg2</code> and so on.</td>
</tr>
</tbody>
</table>

8.6.6 RDF-star extension functions

To avoid any parsing of an embedded triple, GraphDB introduces the following SPARQL functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:boolean rdf:isTriple(variable var)</code></td>
<td>Checks if the variable <code>var</code> is bound to an embedded triple</td>
</tr>
<tr>
<td><code>iri rdf:subject(variable var)</code></td>
<td>Extracts the subject, predicate, or object from a variable bound to an embedded triple</td>
</tr>
<tr>
<td><code>iri rdf:predicate(variable var)</code></td>
<td></td>
</tr>
<tr>
<td><code>rdfTerm rdf:object(variable var)</code></td>
<td></td>
</tr>
<tr>
<td><code>rdf:Statement(iri subj, iri pred, rdfTerm obj)</code></td>
<td>Creates a new embedded statement with the provided values</td>
</tr>
</tbody>
</table>

See more about RDF-star/SPARQL-star syntax [here](#).

8.6.7 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:double geof:distance(geomLiteral geom1, geomLiteral geom2, iri units)</code></td>
<td>Returns the shortest distance in units between any two Points in the two geometric objects as calculated in the spatial reference system of <code>geom1</code>.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>geomLiteral geof:buffer(geomLiteral geom, xsd:double radius, iri units)</code></td>
<td>This function returns a geometric object that represents all Points whose distance from <code>geom1</code> is less than or equal to the radius measured in units. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:convexHull(geomLiteral geom1)</code></td>
<td>This function returns a geometric object that represents all Points in the convex hull of <code>geom1</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:intersection(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>This function returns a geometric object that represents all Points in the intersection of <code>geom1</code> with <code>geom2</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:union(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>This function returns a geometric object that represents all Points in the union of <code>geom1</code> with <code>geom2</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:difference(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>This function returns a geometric object that represents all Points in the set difference of <code>geom1</code> with <code>geom2</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:symDifference(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>This function returns a geometric object that represents all Points in the set symmetric difference of <code>geom1</code> with <code>geom2</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:envelope(geomLiteral geom1)</code></td>
<td>This function returns the minimum bounding box of <code>geom1</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>geomLiteral geof:boundary(geomLiteral geom1)</code></td>
<td>This function returns the closure of the boundary of <code>geom1</code>. Calculations are in the spatial reference system of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>iri geof:getSRID(geomLiteral geom)</code></td>
<td>Returns the spatial reference system URI for <code>geom</code>.</td>
</tr>
<tr>
<td><code>xsd:boolean geof:relate(geomLiteral geom1, geomLiteral geom2, xsd:string matrix)</code></td>
<td>Return true if the spatial relationship between <code>geom1</code> and <code>geom2</code> corresponds to one with 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 <code>geom1</code> is used for spatial calculations.</td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfEquals(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(TFFFFFTFFT)</code></td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfDisjoint(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(FF*FF****)</code></td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfIntersects(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(T******** *T******** ***T****) ***T****)</code></td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfTouches(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(FT******* F**T***** F***T****)</code></td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfCrosses(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(T*T***T**)</code> for P/L, P/A, L/A; <code>(0*T***T**)</code> for L/L</td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfWithin(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(T*F**F**)</code></td>
</tr>
<tr>
<td><code>xsd:boolean geof:sfContains(geomLiteral geom1, geomLiteral geom2)</code></td>
<td>DE-9IM intersection pattern: <code>(T*****FF*)</code></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:boolean geof:sfOverlaps(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TT<strong>T</strong>T**)) for A/A, P/P; ((1<em>T</em><strong>T</strong>)) for L/L.</td>
</tr>
<tr>
<td>xsd:boolean geof:ehEquals(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFFFTFFFT))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehDisjoint(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((FF<em>FF</em>***))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehMeet(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((FT******* F<strong>T</strong>*** F*<strong>T</strong>**))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehOverlap(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((T<em>T</em><strong>T</strong>))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehCovers(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((T<em>TFT</em>FF*))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehCoveredBy(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFF*TFT**))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehInside(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFF*FFT**))</td>
</tr>
<tr>
<td>xsd:boolean geof:ehContains(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((T<em>FF**F</em>))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8eq(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFFFTFFFT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8dc(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((FFTFFTTTT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8ec(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((FFTFTTTTT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8po(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TTTTTTTTT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8tppi(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TTTFTTFFT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8tpp(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFFTTFTTT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8ntpp(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TFFTFFTTT))</td>
</tr>
<tr>
<td>xsd:boolean geof:rcc8ntppi(geomLiteral geom1, geomLiteral geom2)</td>
<td>DE-9IM intersection pattern: ((TTTFFTFFT))</td>
</tr>
</tbody>
</table>
### 8.6.7.1 GeoSPARQL extension functions

On top of the standard GeoSPARQL functions, GraphDB adds a few useful extensions based on the USeekM library. The prefix `ext:` stands for the namespace `<http://rdf.usekm.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 [here](#).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:double ext:area(geomLiteral g)</code></td>
<td>Calculates the area of the surface of the geometry.</td>
</tr>
<tr>
<td><code>geomLiteral ext:closestPoint(geomLiteral g1, geomLiteral g2)</code></td>
<td>For two given geometries, computes the point on the first geometry that is closest to the second geometry.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:containsProperly(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry properly contains the second geometry. <code>Geom1</code> contains properly <code>geom2</code> if all <code>geom1</code> contains <code>geom2</code> and the boundaries of the two geometries do not intersect.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:coveredBy(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry is covered by the second geometry. <code>Geom1</code> is covered by <code>geom2</code> if every point of <code>geom1</code> is a point of <code>geom2</code>.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:covers(geomLiteral g1, geomLiteral g2)</code></td>
<td>Tests if the first geometry covers the second geometry. <code>Geom1</code> covers <code>geom2</code> if every point of <code>geom2</code> is a point of <code>geom1</code>.</td>
</tr>
<tr>
<td><code>xsd:double ext:hausdorffDistance(geomLiteral g1, geomLiteral g2)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>geo:Line ext:shortestLine(geomLiteral g1, geomLiteral g2)</code></td>
<td>Computes the shortest line between two geometries. Returns it as a LineString object.</td>
</tr>
<tr>
<td><code>geomLiteral ext:simplify(geomLiteral g, double d)</code></td>
<td>Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peuker algorithm.</td>
</tr>
<tr>
<td><code>geomLiteral ext:simplifyPreserveTopology(geomLiteral g, double d)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>xsd:boolean ext:isValid(geomLiteral g)</code></td>
<td>Checks whether the input geometry is a valid geometry.</td>
</tr>
</tbody>
</table>

### 8.6.8 Geospatial 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#>`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xsd:double omgeo:distance(numeric lat1, numeric long1, numeric lat2, numeric long2)</code></td>
<td>Computes the distance between two points in kilometers and can be used in <code>FILTER</code> and <code>ORDER BY</code> clauses. Latitude is limited to the range -90 (South) to +90 (North). Longitude is limited to the range -180 (West) to +180 (East).</td>
</tr>
</tbody>
</table>

See more about GraphDB’s geospatial extensions [here](#).
8.7 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-r1).

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.

8.8 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.

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’.

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 conceptualisation 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.
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., 9.11.2 where the major version is 9, the minor version is 11 and the patch version is 2.

Occasional versions may include a modifier after a hyphen, e.g., 10.0.0-RC1 to signal additional information, e.g., 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. New or significantly changed features are released with a higher major or minor version.

GraphDB 10 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.

**9.1 GraphDB 10.0.2**

**Released:** 1 August 2022

**9.1.1 Component versions**

<table>
<thead>
<tr>
<th>RDF4J</th>
<th>Connectors</th>
<th>Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0.3</td>
<td>16.0.1</td>
<td>2.0.2</td>
</tr>
</tbody>
</table>

GraphDB 10.0.2 is a patch release that addresses an issue with migrating existing connectors created in GraphDB 9.x.

If you have already migrated from 9.x to 10.0.1, there is no need to update to 10.0.2.
9.1.2 GraphDB Connectors & Plugins

9.1.2.1 Bug fixing

- GDB-7237 Existing connector with entityFilter from GraphDB 9.x prevents the connector plugin to initialize in GraphDB 10

9.2 GraphDB 10.0.1

Released: 26 July 2022

9.2.1 Component versions

<table>
<thead>
<tr>
<th>RDF4J</th>
<th>Connectors</th>
<th>Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0.3</td>
<td>16.0.0</td>
<td>2.0.2</td>
</tr>
</tbody>
</table>

GraphDB 10.0.1 is a patch release that fixes some of the issues identified in GraphDB 10.0.0, provides minor improvements, and version upgrades of various libraries to address security vulnerabilities.

It is recommended for everyone to upgrade.

9.2.2 GraphDB Engine & Cluster

9.2.2.1 New features and improvements

- GDB-7153 Provide a user-friendly error message when the configured SSL certificate cannot be loaded
- GDB-7132 As a user, I want my license to be checked when creating a cluster

9.2.2.2 Bug fixing

- GDB-7207 Error in implicit cluster transactions leaves stale transactions
- GDB-7184 Out-of-sync node will become follower if verification of entry is synchronous
- GDB-7172 Cluster failed quorum transaction does not always release transaction log lock
- GDB-7171 Named graph not shown by getContexts() when it contains an owl:sameAs statement
- GDB-7165 The external proxy fails when you try using it for a PATCH request to the /rest/cluster/config api
- GDB-7164 The external proxy fails to read the cluster status when GraphDB basic security is enabled
- GDB-7158 Cluster follower does not replicate after out-of-sync
- GDB-7148 Starting a new transaction from the transactions API returns wrong Location header
- GDB-7104 GraphDB fails to restore from snapshot upon initialization
- GDB-7094 Cluster report shows an error in the log file
- GDB-7088 Cannot add a node that was previously removed and has more than one entry in its transaction log
- GDB-7074 Removed cluster nodes cannot be added back in some cases
- GDB-7059 GraphDB no longer allows variables over grouped value as a valid query projection
- GDB-7048 Cannot add a node that was previously unsuccessfully added without a license
• GDB-7033 Cluster node not responsive after all cluster nodes were stopped and some nodes were restarted
• GDB-6952 Restoring a backup more than once may lead to stale cache and some operations not working
• GDB-6755 Rapid cluster topology changes may lead to `OverlappingFileLockException`

9.2.3 GraphDB Workbench

9.2.3.1 Bug fixing

• GDB-7187 Followers should be able to import files from the server
• GDB-7168 Download SPARQL results not working for certain queries
• GDB-7146 Workbench shows stale username after login
• GDB-7145 Workbench login form not showing after expired login
• GDB-7068 When in cluster, a follower Workbench shows the repositories (and locations) of the leader

9.2.4 GraphDB Connectors & Plugins

9.2.4.1 New features and improvements

• GDB-6954 Improved memory usage for the MongoDB connector

9.2.5 GraphDB Distributions & Deployment

9.2.5.1 New features and improvements

• GDB-7191 External proxy returns its internal address with explicit transactions and when deployed with the Helm chart
• GDB-7138 Make the developer examples use remote GraphDB and graphdb-client-api
• GDB-7117 Upgraded RDF4J, Spring, and various other libraries to latest stable versions
• GDB-7110 Removed the obsolete adapter directory from the distribution
• GDB-7109 Removed the obsolete benchmarks directory from the distribution
• GDB-6524 Implement cluster scalability for the Helm chart

9.2.5.2 Bug fixing

• GDB-7161 GraphDB cannot be deployed with enabled security using the Helm chart
• GDB-7160 GraphDB cannot be migrated when security is on and deployed with the Helm chart
9.3 GraphDB 10.0.0

Released: 30 June 2022

9.3.1 Component versions

<table>
<thead>
<tr>
<th>RDF4J</th>
<th>Connectors</th>
<th>Workbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0.2</td>
<td>16.0.0</td>
<td>2.0.0</td>
</tr>
</tbody>
</table>

GraphDB 10.0.0 is the first major release since GraphDB 9.0 was released in September 2019 and introduces numerous improvements and changes that make GraphDB better than ever, including a new cluster implementation.

Important:

**New high-availability cluster** This GraphDB 10 release introduces the new high-availability cluster based on the Raft consensus algorithm. In the new cluster, any node can be either a leader or a follower. The leader is akin to the master node in the old cluster, while a follower is similar to a worker. Unlike the old cluster, each node in the new cluster contains a replica of the data.

The Raft consensus algorithm uses majority voting to determine the current leader, and consensus to confirm every write operation. This ensures a high uptime, zero data loss, fault tolerance, and smooth recovery from unexpected situations and scenarios.

While the old cluster needed to be defined at the repository level, the new cluster is defined per entire GraphDB instance, which means that once a repository is created in the cluster, it automatically becomes part of the cluster.

The new cluster also handles transactions differently, in a manner that is very similar to how transactions are handled in a non-cluster environment. This makes it possible for every transaction to see its own changes before commit and unlocks some new use cases, such as using the Sequences plugin, that were not possible with the old cluster.

**Single distribution and repository type** Unlike previous versions of GraphDB, version 10 is a single distribution that can run in Free, SE, or EE mode, depending on the currently set license. In line with that, we refactored the existing GraphDB repository types (Free, SE, and EE worker) into a single new repository called simply “GraphDB repository”. It is now simpler than ever to create a repository using a GraphDB Free license and upgrade to SE or EE at any time without even a restart.

The single GraphDB distribution requires at least Java 11.

**Licensing and parallelism** We changed how we handle licenses to ensure that you get the optimal performance for the purchased number of cores and we made it possible to export a repository even with an expired license so you never get locked out of your own data.

The graph path search can now run in parallel mode (depending on the number of licensed cores) so that complex path searches run 5 to 10 times faster than before. Parallel mode must be explicitly requested as part of the query. This is just one of the optimizations we are working on in order to let customers who purchased a larger number of cores benefit not only from handling a big number of concurrent requests, but also get advantage in terms of faster processing of a single complex query.

**Connector filtering redesign** We listened to our users and did a major redesign of the connector filtering mechanism to allow for very fine-tuned filtering at every level of the indexing process, and we added two-variable comparison to cover numerous new use cases that were not possible with the old filtering. If you use connector filters, we urge you to get acquainted with the changes now to make sure you can benefit from them when you upgrade to the final GraphDB 10 release.

**Workbench improvements** We have retouched the GraphDB Workbench to make it even easier to use and we added an experimental dark theme (accessible via Setup  My settings).

GraphDB 10 also introduces a French translation of the Workbench interface. You can change the language from the language selection button in the top right corner.
Remote locations refactored Remote locations have been simplified. They now serve mainly as a facility for easier cluster management. Remote locations in GraphDB 10 cannot be activated but the repositories from them are accessible from the Workbench together with the local repositories.

Upgraded to the latest RDF4J 4.0 RDF4J 4 introduces Java 11 as a minimum requirement, just like GraphDB 10. It includes numerous fixes and improvements that improve the overall performance.

OntoRefine removed from GraphDB OntoRefine has been removed from GraphDB and is now developed as a separate product, part of our line of tools for data transformation. All existing OntoRefine and RDF mapping functionality will remain available in the new separate product.

Warning: The new release of OntoRefine is expected around 25 July 2022, and the OntoRefine CLI about a month later. If you need to use OntoRefine now, please hold off your upgrade from GraphDB 9.11 to 10 for a while.

9.3.2 GraphDB Engine & Cluster

9.3.2.1 New features and improvements

- GDB-6904 As a user, I need to make a full GraphDB instance backup and restore through a REST API
- GDB-6903 Support Java 17 for running GraphDB
- GDB-6888 As a user, I want to use IPv6 with GraphDB out of the box
- GDB-6849 Implement embedded load balancer/proxy
- GDB-6818 Implement external load balancer
- GDB-6762 As a user, I want to load balance Boolean & split queries in a cluster
- GDB-6737 As a user, I want to be able to replicate SQL Views in a cluster
- GDB-6726 Multiple cases of unclosed iterators/connections in both production and test code
- GDB-6684 As a user, I want to be able to back up and restore an entire GraphDB instance
- GDB-6683 Make the storage tool more user-friendly when scanning the predicate statistics
- GDB-6679 As an admin, I want user changes to be replicated in a cluster
- GDB-6677 As a user, I need to update the cluster configuration parameters with a PATCH request
- GDB-6664 As a user, I want to know all GraphDB HTTP addresses for a cluster group and identify the leader node
- GDB-6655 Remove OntoRefine from the project and the distribution
- GDB-6602 High-availability Cluster load balancing
- GDB-6575 Upgrade versions of various libraries
- GDB-6541 As a user, I want to be able to secure my GraphDB cluster group
- GDB-6540 As a user, I want to verify the GraphDB nodes state when joining cluster group
- GDB-6539 As a user, I need to receive a response with an error message when I try to create an invalid cluster configuration
- GDB-6533 As a security administrator, I need to provide case-insensitive user names for user accounts
- GDB-6521 As a DevOps, I want to be able to deploy GraphDB 10.0 using Helm
- GDB-6494 As a user of GraphDB REST API, I need it to be intuitive, consistent, and conform to the best practices
- GDB-6456 Migrate GraphDB to RDF4J 4.0.2
• GDB-6407 Remove Jolokia and replace its uses with a REST API
• GDB-6399 As a user, I want to add/remove node in a running cluster group though a REST endpoint
• GDB-6372 Introduce automatic snapshot replication for out-of-sync followers
• GDB-6336 Redesign command line tools to be more user-friendly and straightforward
• GDB-6274 As a user, I want to use a Java client that would provide high availability of cluster
• GDB-6268 As a user, I want to have cluster node defined per instance
• GDB-6267 As a user, I want to have runtime group membership management
• GDB-6266 As a user, I want to have automatic query load balancing
• GDB-6265 As a user, I want to have automatic log replication between server nodes
• GDB-6264 As a user, I want to have automatic leader election for fault tolerance
• GDB-6236 As a user, I want to export my repository when the license is expired
• GDB-6235 As a user, I want setting a new license to take effect without a restart
• GDB-6189 Optimized resource closure and disposal
• GDB-6183 As a user, I want the number of licensed cores to be used optimally
• GDB-6055 As a user, I want to take advantage of parallel graph path query evaluation
• GDB-5548 Allow shapes to be stored in a user specified named graph

9.3.2.2 Bug fixing

• GDB-6777 Some easily processable Elasticsearch aggregations are not retrieved by the Elasticsearch connector
• GDB-6770 Incorrect content type on some of the SPARQL template endpoints
• GDB-6756 Bound variable for SERVICE endpoint not visible
• GDB-6623 Tx-rollback-plugin packaged as a dependency of GraphDB
• GDB-6582 License information is logged multiple times even when it has not changed
• GDB-6544 Confusing warning when reverting to Free
• GDB-6410 Query optimizer incorrectly selects a statement pattern with all unbound variables after arbitrary length property path

9.3.3 GraphDB Workbench

9.3.3.1 New features and improvements

• GDB-6671 As a user, I want to create/edit/delete the cluster group in the Workbench
• GDB-6517 As a user, I need a new styled and better looking Workbench
• GDB-6493 As a user, I need to use a remote location only to manage external GraphDB
• GDB-6401 As a user, I want to see the cluster state in the Workbench
• GDB-6273 As a user I want to configure and view the state of the cluster group through the Workbench
• GDB-5345 Internationalization support in the Workbench and preliminary French translation
• GDB-4288 As a user, I need to explore nested triples in the Visual graph
9.3.3.2 Bug fixing

- GDB-6946 GraphDB unable to find base URL when the URL is over 4,096 symbols
- GDB-6732 There is no validation for the prefix and namespace IRI during namespace create/update
- GDB-6462 Visual graph does not show relations direction in some cases
- GDB-6225 When importing a directory from Import Server files, the status of the content of the directory is not updated

9.3.4 GraphDB Connectors & Plugins

9.3.4.1 New features and improvements

- GDB-6772 Add cluster support to Sequences plugin
- GDB-6744 Extend plugin API with support for adding/removing statements in a safe manner
- GDB-6486 Introduce graph() pseudo URI that allows indexing the named graph of a connector field value
- GDB-6487 Introduce the ability to start a connector nested document from one of the parents instead of the actual root of the nested document
- GDB-5419 As a user, I need connector filtering with two independent variables
- GDB-6136 As a user, I need to filter values and documents separately when I create a connector

9.3.4.2 Bug fixing

- GDB-6742 Text Mining plugin fails to annotate document
- GDB-6555 Nested documents merged incorrectly when using multiple property chain variants
- GDB-3616 GraphDB JavaScript driver cannot upload .ttl files larger than 10 megabytes

9.3.5 GraphDB Distributions & Deployment

9.3.5.1 New features and improvements

- GDB-3641 As a user, I want to use a single GraphDB distribution with access to features based on my license
- GDB-6181 Require OpenJDK 11 as a minimal version
- GDB-6247 As a user, I want to use one repository type regardless of the license
10.1 General

10.1.1 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.

10.1.2 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.

10.1.3 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.

10.2 Configuration

10.2.1 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:

```
```

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.
10.2.2 What is a repository?

A repository is essentially a single GraphDB database. Multiple repositories can be active at the same time and they are isolated from each other.

10.2.3 How do I create a repository?

Go to Setup  Repositories, and follow the instructions.

10.2.4 How do I retrieve repository configurations?

To see what configuration data is stored in a GraphDB repository, go to Repositories and use the Download repository configuration as Turtle icon.

Then open the result file named repositoryname-config.ttl, which contains this information.

10.2.5 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.

10.2.6 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.

10.2.7 How do I create a GraphDB EE cluster without knowing JMX?

Create some master and worker repositories first (in a production cluster each master and worker should be in a separate GraphDB instance). Go to Setup  Cluster management, where you will see a visual representation of repositories and each cluster. Drag and drop workers onto masters to connect them.
10.3 RDF & SPARQL

10.3.1 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.

10.3.2 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.

10.3.3 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.

10.4 Security

10.4.1 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.
10.4.2 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).

10.5 Troubleshooting

10.5.1 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).

10.5.2 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.
SUPPORT

- **email**: graphdb-support@ontotext.com
- **Twitter**: @OntotextGraphDB
- **GraphDB tag on Stack Overflow** at http://stackoverflow.com/questions/tagged/graphdb