
GraphDB SE Documentation

Release 9.0

Ontotext

Oct 03, 2019

CONTENTS

1	General	1
1.1	About GraphDB	2
1.2	Architecture & Components	2
1.2.1	Architecture	2
1.2.1.1	RDF4J	3
1.2.1.2	The Sail API	4
1.2.2	Components	4
1.2.2.1	Engine	4
1.2.2.2	Connectors	5
1.2.2.3	Workbench	5
1.3	GraphDB SE	5
1.3.1	Comparison of GraphDB Free and GraphDB SE	6
1.4	Connectors	6
1.5	Workbench	6
2	Quick start guide	9
2.1	Run GraphDB as a Standalone Server	9
2.1.1	Running GraphDB	10
2.1.1.1	Options	10
2.1.2	Configuring GraphDB	10
2.1.2.1	Paths and network settings	10
2.1.2.2	Java virtual machine settings	11
2.1.3	Stopping the database	11
2.2	Set up Your License	12
2.3	Create a Repository	13
2.4	Load Your Data	14
2.4.1	Load data through the GraphDB Workbench	14
2.4.2	Load data through SPARQL or RDF4J API	16
2.4.3	Load data through the GraphDB LoadRDF tool	16
2.5	Explore Your Data and Class Relationships	16
2.5.1	Explore instances	16
2.5.2	Create your own visual graph	20
2.5.3	Class hierarchy	21
2.5.4	Domain-Range graph	23
2.5.5	Class relationships	25
2.6	Query Your Data	28
2.6.1	Query data through the Workbench	28
2.6.2	Query data programmatically	32
2.7	Additional Resources	32
3	Installation	33
3.1	Requirements	33
3.1.1	Minimum requirements	33
3.1.2	Hardware sizing	33

3.1.3	Licensing	34
3.2	Running GraphDB	34
3.2.1	Run GraphDB as a Standalone Server	34
3.2.1.1	Running GraphDB	34
3.2.1.2	Configuring GraphDB	35
3.2.1.3	Stopping the database	36
3.3	Configuring GraphDB	36
3.3.1	Directories	37
3.3.1.1	GraphDB Home	37
3.3.1.2	Checking the configured directories	38
3.3.2	Configuration	38
3.3.2.1	Config properties	38
3.3.2.2	Configuring logging	39
3.3.3	Best practices	40
3.3.3.1	Step by step guide	40
3.4	Migrating GraphDB Configurations	40
3.4.1	Compatibility between the versions of GraphDB, Connectors and third party connectors	41
3.5	Distribution package	41
3.6	Using Maven Artifacts	42
3.6.1	Public Maven repository	42
3.6.2	Distribution	43
3.6.3	GraphDB JAR file for embedding the database or plugin development	43
3.6.4	I want to proxy the repository in my nexus	44
4	Administration	45
4.1	Administration tasks	45
4.2	Administration Tools	45
4.2.1	Workbench	46
4.2.2	JMX interface	46
4.2.2.1	Configuring the JMX endpoint	46
4.3	Setting up Licenses	46
4.3.1	Setting up licenses through the Workbench	47
4.3.2	Setting up licenses through a file	48
4.3.2.1	Custom file path property	48
4.3.3	Order of preference	49
4.3.4	Deprecated methods	49
4.4	Creating Locations	49
4.4.1	Active location	49
4.4.2	Inactive location	52
4.4.3	Connect to a remote location	52
4.4.4	Configure a data location	53
4.5	Creating a Repository	54
4.5.1	Create a repository	54
4.5.1.1	Using the Workbench	54
4.5.1.2	Using the RDF4J console	55
4.5.2	Manage repositories	55
4.5.2.1	Select a repository	55
4.5.2.2	Make it a default repository	55
4.5.2.3	Edit a repository	56
4.6	Configuring a Repository	56
4.6.1	Plan a repository configuration	56
4.6.2	Configure a repository through the GraphDB Workbench	57
4.6.3	Edit a repository	57
4.6.4	Configure a repository programmatically	58
4.6.5	Configuration parameters	59
4.6.6	Configure GraphDB memory	62
4.6.6.1	Configure Java heap memory	62
4.6.6.2	Single global page cache	62

4.6.6.3	Configure Entity pool memory	62
4.6.6.4	Sample memory configuration	63
4.6.6.5	Upper bounds for the memory consumed by the GraphDB process	63
4.6.7	Reconfigure a repository	63
4.6.7.1	Using the Workbench	63
4.6.7.2	Global overrides	63
4.6.8	Rename a repository	64
4.6.8.1	Using the workbench	64
4.7	Secure GraphDB	64
4.7.1	Enable security	64
4.7.2	Login and default credentials	65
4.7.3	Free access	65
4.7.4	Users and Roles	66
4.7.4.1	Create new user	66
4.7.4.2	Set password	67
4.8	Request Tracking	67
4.9	Application Settings	67
4.10	Backing up and Restoring a Repository	68
4.10.1	Back up a repository	68
4.10.1.1	Export repository to an RDF file	69
4.10.1.2	Back up a repository using JMX interface	70
4.10.1.3	Back up GraphDB by copying the binary image	72
4.10.2	Restore a repository	72
4.11	Query Monitoring and Termination	73
4.11.1	Query monitoring and termination using the workbench	73
4.11.2	Query monitoring and termination using the JMX interface	74
4.11.2.1	Query monitoring	74
4.11.2.2	Terminating a query	75
4.11.3	Terminating a transaction	75
4.11.4	Automatically prevent long running queries	76
4.12	Performance Optimizations	76
4.12.1	Data loading & query optimizations	76
4.12.1.1	Dataset loading	77
4.12.1.2	GraphDB's optional indices	78
4.12.1.3	Cache/index monitoring and optimizations	78
4.12.1.4	Query optimizations	79
4.12.2	Explain Plan	80
4.12.2.1	What is GraphDB's Explain Plan	80
4.12.2.2	Activating the explain plan	81
4.12.2.3	Simple explain plan	81
4.12.2.4	Multiple triple patterns	82
4.12.2.5	Wine queries	82
4.12.3	Inference optimizations	84
4.12.3.1	Delete optimizations	84
4.12.3.2	Rules optimizations	86
4.12.3.3	Optimization of owl:sameAs	95
4.12.3.4	RDFS and OWL support optimizations	101
4.13	Troubleshooting	102
4.13.1	Database health checks	102
4.13.1.1	Possible values for health checks and their meaning	103
4.13.1.2	Default health checks for the different GraphDB editions	103
4.13.1.3	Aggregated health checks	103
4.13.1.4	Running legacy health checks	105
4.13.2	System metrics monitoring	105
4.13.2.1	Page cache metrics	106
4.13.2.2	Entity pool metrics	106
4.13.3	Diagnosing and reporting critical errors	106
4.13.3.1	Report	107

4.13.3.2	Logs	108
4.13.4	Storage tool	109
4.13.4.1	Options	109
4.13.4.2	Supported commands	110
4.13.4.3	Examples	110
5	Usage	113
5.1	Loading Data	113
5.1.1	Loading data using the Workbench	113
5.1.1.1	Import settings	114
5.1.1.2	Importing local files	115
5.1.1.3	Importing remote content	116
5.1.1.4	Importing RDF data from a text snippet	116
5.1.1.5	Importing server files	117
5.1.1.6	Import data with an INSERT query	117
5.1.2	Loading data using the LoadRDF tool	117
5.1.2.1	Command Line Options	118
5.1.2.2	A GraphDB Repository Configuration Sample	119
5.1.2.3	Tuning LoadRDF	120
5.1.3	Loading data using the Preload tool	120
5.1.3.1	Preload vs LoadRDF	120
5.1.3.2	Command line option	121
5.1.3.3	A GraphDB repository configuration sample	122
5.1.3.4	Tuning Preload	122
5.1.3.5	Resuming data loading with Preload	122
5.1.4	Loading data using OntoRefine	123
5.1.4.1	OntoRefine - overview and features	123
5.1.4.2	Example Data	123
5.1.4.3	Upload data in OntoRefine	124
5.1.4.4	Viewing tabular data as RDF	126
5.1.4.5	RDFizing data	128
5.1.4.6	Importing data in GraphDB	129
5.1.4.7	Rows and records	131
5.1.4.8	Additional resources	138
5.2	Exploring data	138
5.2.1	Class hierarchy	138
5.2.1.1	Explore your data - different actions	139
5.2.1.2	Domain-range graph	142
5.2.2	Class relationships	142
5.2.3	Explore resources	146
5.2.3.1	Explore resources through the easy graph	146
5.2.3.2	Create your own visual graph	154
5.2.3.3	Saved graphs	156
5.2.4	Viewing and editing resources	156
5.2.4.1	View and add a resource	156
5.2.4.2	Edit a resource	158
5.3	Querying Data	158
5.3.1	Save and share queries	160
5.3.2	Interrupt queries	161
5.4	Exporting Data	161
5.4.1	Exporting a repository	162
5.4.2	Exporting individual graphs	162
5.4.3	Exporting query results	163
5.4.4	Exporting resources	163
5.5	Reasoning	163
5.5.1	Logical formalism	164
5.5.2	Rule format and semantics	164
5.5.3	The ruleset file	165

5.5.3.1	Prefices	165
5.5.3.2	Axioms	165
5.5.3.3	Rules	166
5.5.4	Rulesets	169
5.5.4.1	Predefined rulesets	169
5.5.4.2	Custom rulesets	170
5.5.5	Inference	171
5.5.5.1	Reasoner	171
5.5.5.2	Rulesets execution	171
5.5.5.3	Retraction of assertions	172
5.5.6	How To's	173
5.5.6.1	Operations on rulesets	173
5.5.6.2	Reinferring	176
5.6	Using the Workbench REST API	176
5.6.1	Security management	177
5.6.2	Location management	177
5.6.3	Repository management	177
5.6.4	Cluster management	177
5.6.5	Data import	177
5.6.6	Saved queries	177
5.7	Using GraphDB with the RDF4J API	178
5.7.1	RDF4J API	179
5.7.1.1	Accessing a local repository	179
5.7.1.2	Accessing a remote repository	180
5.7.2	SPARQL endpoint	180
5.7.3	Graph Store HTTP Protocol	181
5.8	Plugins	181
5.8.1	Plugin API	181
5.8.1.1	What is the GraphDB Plugin API	182
5.8.1.2	Description of a GraphDB plugin	182
5.8.1.3	The life cycle of a plugin	184
5.8.1.4	Repository internals	185
5.8.1.5	Query processing	190
5.8.1.6	Update processing	195
5.8.1.7	Transactions	198
5.8.1.8	Exceptions	199
5.8.1.9	Accessing other plugins	199
5.8.1.10	List of plugin interfaces and classes	200
5.8.1.11	Putting it all together: example plugins	205
5.8.2	RDF rank	210
5.8.2.1	What is RDF Rank	210
5.8.2.2	Parameters	211
5.8.2.3	Full computation	212
5.8.2.4	Incremental updates	213
5.8.2.5	Exporting RDF Rank values	213
5.8.2.6	Checking the RDF Rank status	213
5.8.2.7	Rank Filtering	214
5.8.3	Semantic similarity searches	215
5.8.3.1	Why do I need the similarity plugin?	216
5.8.3.2	What the similarity plugin does?	216
5.8.3.3	How the similarity plugin works?	216
5.8.3.4	Download data	222
5.8.3.5	Text based similarity searches	224
5.8.3.6	Predication-based Semantic Indexing	232
5.8.3.7	Hybrid indexing	236
5.8.3.8	Training cycles	239
5.8.4	JavaScript functions	239
5.8.4.1	How to register JS function	239

	5.8.4.2	How to remove JS function	240
5.9		User-defined Indexes	241
	5.9.1	Autocomplete index	241
	5.9.1.1	Autocomplete in the SPARQL editor	242
	5.9.1.2	Autocomplete in the View resource on Home page	243
	5.9.2	GeoSPARQL support	243
	5.9.2.1	What is GeoSPARQL	244
	5.9.2.2	Usage	245
	5.9.2.3	GeoSPARQL examples	248
	5.9.3	Geo-spatial extensions	252
	5.9.3.1	What are geo-spatial extensions	253
	5.9.3.2	How to create a geo-spatial index	253
	5.9.3.3	Geo-spatial query syntax	254
	5.9.3.4	Extension query functions	256
	5.9.3.5	Implementation details	257
	5.9.4	Lucene Full-text Search	258
	5.9.4.1	RDF search	258
5.10		GraphDB connectors	266
	5.10.1	Lucene GraphDB connector	266
	5.10.1.1	Overview and features	268
	5.10.1.2	Usage	268
	5.10.1.3	Setup and maintenance	270
	5.10.1.4	Working with data	275
	5.10.1.5	List of creation parameters	279
	5.10.1.6	Datatype mapping	284
	5.10.1.7	Advanced filtering and fine tuning	284
	5.10.1.8	Overview of connector predicates	289
	5.10.1.9	Caveats	290
	5.10.1.10	Upgrading from previous versions	290
5.11		MongoDB Integration	291
	5.11.1	Overview and features	292
	5.11.2	Usage	292
	5.11.3	Setup and maintenance	295
	5.11.3.1	Installing MongoDB	295
	5.11.3.2	Creating an index	295
	5.11.3.3	Upgrading an index	296
	5.11.3.4	Deleting an index	296
	5.11.3.5	Loading sample data	296
	5.11.3.6	Querying MongoDB	296
	5.11.3.7	Multiple index calls in the same query	297
	5.11.3.8	Using aggregation functions	299
	5.11.3.9	Custom fields	299
	5.11.3.10	Authentication	300
5.12		Internal SPARQL federation	300
	5.12.1	Usage	300
	5.12.2	Parameters	301
5.13		GraphDB dev guide	301
	5.13.1	Storage	301
	5.13.1.1	What is GraphDB's persistence strategy	302
	5.13.1.2	GraphDB's indexing options	302
	5.13.2	Notifications	305
	5.13.2.1	What are GraphDB local notifications	305
	5.13.2.2	What are GraphDB remote notifications	306
	5.13.3	Query behaviour	308
	5.13.3.1	What are named graphs	308
	5.13.3.2	How to manage explicit and implicit statements	309
	5.13.3.3	How to query explicit and implicit statements	310
	5.13.3.4	How to specify the dataset programmatically	311

5.13.3.5	How to access internal identifiers for entities	311
5.13.3.6	How to use RDF4J ‘direct hierarchy’ vocabulary	313
5.13.3.7	Other special GraphDB query behaviour	313
5.13.4	Retain BIND position special graph	313
5.13.5	Graph Replacement Optimization	315
5.14	Experimental features	315
5.14.1	SPARQL-MM support	315
5.14.1.1	Usage examples	316
5.14.2	Provenance plugin	318
5.14.2.1	When to use the Provenance plugin	318
5.14.2.2	Predicates	318
5.14.2.3	Enabling the plugin	318
5.14.2.4	Using the plugin - examples	319
5.14.3	Nested repositories	320
5.14.3.1	What are nested repositories	321
5.14.3.2	Inference, indexing and queries	322
5.14.3.3	Configuration	322
5.14.3.4	Initialisation and shut down	322
5.14.4	LVM-based backup and replication	323
5.14.4.1	Prerequisites	323
5.14.4.2	How it works	323
5.14.4.3	Some further notes	324
6	Security	325
6.1	Authorization	325
6.1.1	User roles and permissions	325
6.1.2	Built-in users and roles	326
6.2	Authentication	326
6.2.1	Local	327
6.2.2	LDAP	328
6.2.3	Basic Authentication	329
6.3	Encryption	330
6.3.1	Encryption in transit	330
6.3.1.1	Enable SSL/TLS	330
6.3.1.2	HTTPS in the Cluster	330
6.3.2	Encryption at rest	331
6.4	Security auditing	331
7	Developer Hub	333
7.1	Data modelling with RDF(S)	333
7.1.1	What is RDF?	333
7.1.2	What is RDFS?	334
7.1.3	See also	335
7.2	SPARQL	335
7.2.1	What is SPARQL?	335
7.2.2	Using SPARQL in GraphDB	337
7.2.3	See also	337
7.3	Ontologies	338
7.3.1	What is an ontology?	338
7.3.2	What are the benefits of developing and using an ontology?	338
7.3.3	Using ontologies in GraphDB	339
7.3.4	See also	339
7.4	Inference	339
7.4.1	What is inference?	339
7.4.2	Inference in GraphDB	339
7.4.2.1	Standard rule-sets	340
7.4.2.2	Custom rule-sets	340
7.5	Programming with GraphDB	341

7.5.1	Installing Maven dependencies	341
7.5.2	Examples	341
7.5.2.1	Hello world in GraphDB	341
7.5.2.2	Family relations app	343
7.5.2.3	Embedded GraphDB	346
7.6	Extending GraphDB Workbench	349
7.6.1	Clone, download, and run GraphDB Workbench	350
7.6.2	Add your own page and controller	350
7.6.3	Add repository checks	351
7.6.4	Repository setup	352
7.6.5	Select departure and destination airport	352
7.6.6	Find the paths between the selected airports	353
7.6.7	Visualize results	359
7.6.8	Add status message	359
7.7	Workbench REST API	359
7.7.1	Repository management with the Workbench REST API	359
7.7.1.1	Prerequisites	360
7.7.1.2	Managing repositories	360
7.7.1.3	Managing locations	361
7.7.1.4	Further reading	362
7.7.2	Cluster management with the Workbench REST API	362
7.7.2.1	Prerequisites	363
7.7.2.2	Creating a cluster	363
7.7.2.3	Further reading	365
7.7.3	Workbench REST API curl commands	365
7.7.3.1	Location Management	366
7.7.3.2	Repository Management	367
7.7.3.3	Data import	373
7.7.3.4	Saved Queries	375
7.7.3.5	Cluster Management	376
7.8	Visualize GraphDB data with Ogma JS	378
7.8.1	Related to google people and organization in factforge.net	378
7.8.2	Suspicious control chain through off-shore company in factforge.net	381
7.8.3	Flights shortest path	384
7.8.4	Common function to visualize GraphDB Data	388
7.9	Create custom graph view over your RDF data	389
7.9.1	How it works?	390
7.9.2	World airport, airline and route data	390
7.9.2.1	Data model	391
7.9.2.2	Configured queries	391
7.9.3	Springer Nature SciGraph	393
7.9.3.1	Data model	393
7.10	GraphDB System Statements	397
7.10.1	System graphs	397
7.10.2	System predicates	398
7.11	Time Functions Extensions	398
7.11.1	Period extraction functions	398
7.11.2	Period transformation functions	399
7.11.3	Durations Expressed in Certain Units	400
7.11.4	Arithmetic operations	400
7.12	GraphDB Fundamentals	400
7.12.1	Module 1: RDF & RDFS	400
7.12.2	Module 2: SPARQL	401
7.12.3	Module 3: Ontology	401
7.12.4	Module 4: GraphDB Installation	401
7.12.5	Module 5: Performance Tuning & Scalability	401
7.12.6	Module 6: GraphDB Workbench & RDF4J	401
7.12.7	Module 7: Loading Data	401

7.12.8	Module 8: Rule Set & Reasoning Strategies	401
7.12.9	Module 9: Extensions	401
7.12.10	Module 10: Troubleshooting	402
7.13	FAQ	402
8	References	403
8.1	Introduction to the Semantic Web	403
8.1.1	Resource Description Framework (RDF)	404
8.1.1.1	Uniform Resource Identifiers (URIs)	405
8.1.1.2	Statements: Subject-Predicate-Object Triples	405
8.1.1.3	Properties	407
8.1.1.4	Named graphs	407
8.1.2	RDF Schema (RDFS)	408
8.1.2.1	Describing classes	408
8.1.2.2	Describing properties	408
8.1.2.3	Sharing vocabularies	409
8.1.2.4	Dublin Core Metadata Initiative	410
8.1.3	Ontologies and knowledge bases	410
8.1.3.1	Classification of ontologies	411
8.1.3.2	Knowledge bases	412
8.1.4	Logic and inference	412
8.1.4.1	Logic programming	412
8.1.4.2	Predicate logic	413
8.1.4.3	Description logic	414
8.1.5	The Web Ontology Language (OWL) and its dialects	414
8.1.5.1	OWL DLP	416
8.1.5.2	OWL Horst	416
8.1.5.3	OWL2 RL	417
8.1.5.4	OWL Lite	417
8.1.5.5	OWL DL	417
8.1.6	Query languages	417
8.1.6.1	RQL, RDQL	417
8.1.6.2	SPARQL	418
8.1.6.3	SeRQL	418
8.1.7	Reasoning strategies	418
8.1.7.1	Total materialization	419
8.1.8	Semantic repositories	419
8.2	GraphDB feature comparison	420
8.3	Repository configuration template - how it works	420
8.4	Ontology mapping with owl:sameAs property	421
8.5	Workbench User Interface	423
8.5.1	Workbench Functionalities Descriptions	426
8.5.2	Workbench configuration properties	427
8.6	SPARQL compliance	427
8.6.1	SPARQL 1.1 Protocol for RDF	427
8.6.2	SPARQL 1.1 Query	428
8.6.3	SPARQL 1.1 Update	428
8.6.3.1	Modification operations on the RDF triples:	428
8.6.3.2	Operations for managing graphs:	428
8.6.4	SPARQL 1.1 Federation	428
8.6.4.1	Internal SPARQL federation	429
8.6.5	SPARQL 1.1 Graph Store HTTP Protocol	430
8.6.5.1	URL patterns for this new functionality are provided at:	430
8.6.5.2	Methods supported by these resources and their effects:	430
8.6.5.3	Request headers:	430
8.6.5.4	Supported parameters for requests on indirectly referenced named graphs:	430
8.7	Using math functions with SPARQL	431
8.8	OWL compliance	434

8.9	Glossary	435
9	Release notes	437
9.1	GraphDB 9.0.0	437
9.1.1	Component versions	437
9.1.2	GraphDB Engine	438
9.1.2.1	New features	438
9.1.2.2	Bug fixing	438
9.1.3	GraphDB Workbench	439
9.1.3.1	New features	439
9.1.3.2	Bug fixing	439
9.1.4	GraphDB Plugins	439
9.1.4.1	New features	439
9.1.4.2	Bug fixing	439
9.1.5	GraphDB Connectors	440
9.1.5.1	New features	440
9.1.5.2	Bug fixing	440
9.1.6	GraphDB Cluster	440
9.1.6.1	Bug fixing	440
10	FAQ	441
11	Support	443

GENERAL

Hint: This documentation is written to be used by technical people. Whether you are a database engineer or system designer evaluating how this database fits to your system, or you are a developer who has already integrated it and actively employs its power - this is the complete reference. It is also useful for system administrators who need to support and maintain a GraphDB-based system.

Note: The GraphDB documentation presumes that the reader is familiar with databases. The required minimum of Semantic Web concepts and related information is provided in the *Introduction to the Semantic Web* section in *References*.

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.

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*).

The development of GraphDB is partly supported by *SEKT*, *TAO*, *TripCom*, *LarKC*, and other *FP6* and *FP7* European research projects.

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: Tips badges are handy pieces of information.

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

Warning: Warnings are pieces of advice that turn your attention to things you should be cautious about.

1.1 About GraphDB

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 triple stores 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.

Ontotext offers three editions of GraphDB: Free, Standard and Enterprise.

- **GraphDB Free** - commercial, file-based, sameAs & query optimisations, scales to tens of billions of RDF statements on a single server with a limit of two concurrent queries;
- **GraphDB Standard Edition (SE)** - commercial, file-based, sameAs & query optimisations, scales to tens of billions of RDF statements on a single server and an unlimited number of concurrent queries;
- **GraphDB Enterprise Edition (EE)** - high-availability cluster with worker and master database implementation for resilience and high-performance parallel query answering.

To find out more about the differences between the editions, see the *GraphDB feature comparison* section.

1.2 Architecture & Components

What's in this document?

- *Architecture*
 - *RDF4J*
 - *The Sail API*
- *Components*
 - *Engine*
 - *Connectors*
 - *Workbench*

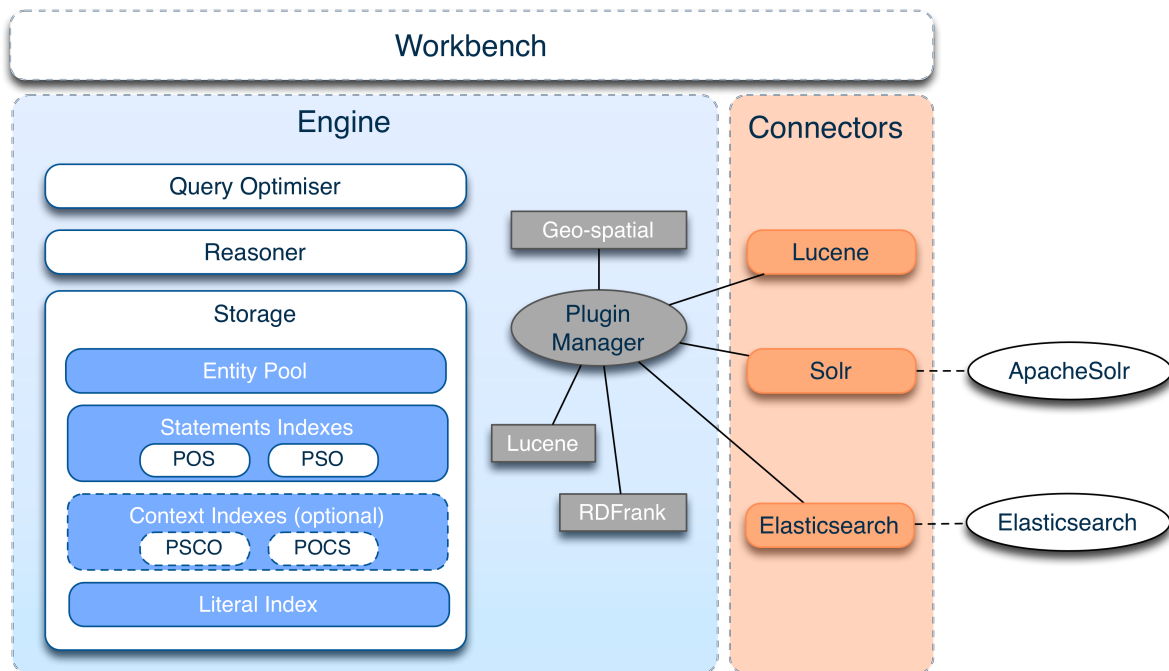
1.2.1 Architecture

GraphDB is packaged as a Storage And Inference Layer (SAIL) 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 optimised 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.



GraphDB High-level Architecture

1.2.1.1 RDF4J

The **RDF4J framework** is a framework for storing, querying, and reasoning with RDF data. It is implemented in Java by Aduna as an open source project and includes various storage back-ends (memory, file, database), query languages, reasoners, and client-server protocols.

There are essentially two ways to use RDF4J:

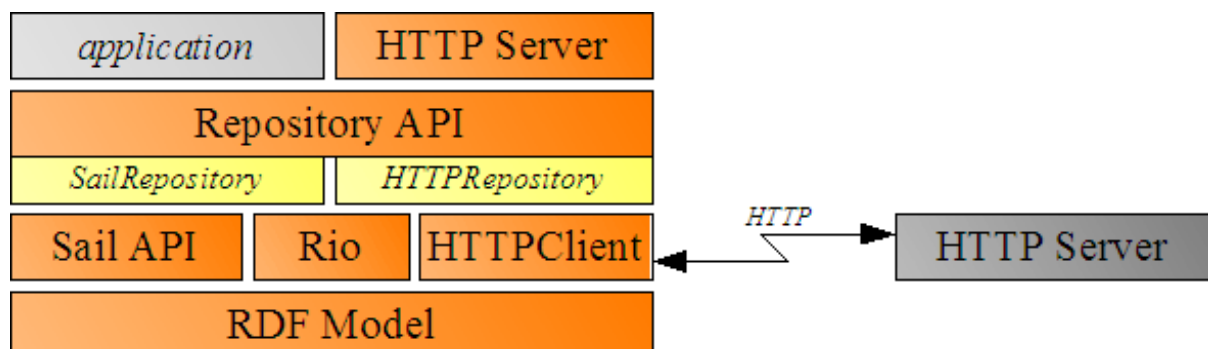
- as a standalone server;
- embedded in an application as a Java library.

RDF4J supports the W3C SPARQL query language, as well as the most popular RDF file formats and query result formats.

RDF4J offers a JDBC-like user API, streamlined system APIs and a RESTful HTTP interface. Various extensions are available or are being developed by third parties.

RDF4J Architecture

The following is a schematic representation of the RDF4J architecture and a brief overview of the main components.



The RDF4J architecture

The RDF4J framework is a loosely coupled set of components, where alternative implementations can be easily exchanged. RDF4J comes with a variety of Storage And Inference Layer (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 indices on statements, POS and PSO, *context index* CPSO, 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 and literals) 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*

1.2.2.3 Workbench

The *Workbench* is the GraphDB web-based administration tool.

1.3 GraphDB SE

What's in this document?

- *Comparison of GraphDB Free and GraphDB SE*

What makes GraphDB SE different?

- Manage **10's of billions of RDF statements** on a single server;
- Performs query and reasoning operations using **file-based indices**;
- 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 optimised rulesets*;
- Optimised support for data integration through *owl:sameAs*;
- Special indices for *efficient geo-spatial constraints* (near-by, within, distance);
- *Full-text search*, based on Lucene;
- 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 optimiser* allowing for the evaluation of different query plans;
- *RDF rank* to order query results by relevance or other measures;
- *Notification* allowing clients to react to statements in the update stream;
- *Lucene connector* for extremely fast normal and faceted (aggregation) searches; automatically stays up-to-date with the GraphDB data;
- *GraphDB Workbench* - the default web-based administration tool;

- *LoadRDF* for very fast repository creation from big datasets;

GraphDB Standard (SE) is the commercial standalone edition of GraphDB. It is implemented in Java and packaged as a Storage and Inference Layer (SAIL) for the *RDF4J* RDF framework. GraphDB SE is a native RDF rule-entailment and storage engine. The supported semantics can be configured through ruleset definition and selection. Included are *rulesets* for OWL-Horst, unconstrained RDFS with OWL Lite and the OWL2 profiles RL and QL. *Custom rulesets* allow tuning for *optimal performance* and expressivity.

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

1.3.1 Comparison of GraphDB Free and GraphDB SE

GraphDB SE and GraphDB Free – are identical in terms of usage and integration and share most features:

- designed as an enterprise-grade semantic repository system;
- suitable for massive volumes of data;
- file-based indices (enables it to scale to billions of statements even on desktop machines);
- *inference* and *query optimisations* (ensures fast query evaluations).

GraphDB SE

- suitable for heavy query loads.

GraphDB Free

- suitable for low query loads and smaller projects.

1.4 Connectors

The GraphDB 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.

The 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 terms of *RDF*, this corresponds to a set of triples that have the same subject. In addition to simple properties (defined by a single triple), the Connectors support property chains. A property chain is a sequence of triples where each triple's object is the subject of the subsequent triple.

GraphDB SE comes with the following connector implementations:

- *Lucene 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*;

- Import of server files;
- Export in more formats;
- Query monitoring with the possibility to kill a long running query;
- System resource monitoring;
- User and permission management;
- Connector management;
- Cluster management.

The 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;
- provides REST API for automating various tasks for managing and administering 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](#).

QUICK START GUIDE

What's in this document?

- *Run GraphDB as a Standalone Server*
 - *Running GraphDB*
 - *Configuring GraphDB*
 - *Stopping the database*
- *Set up Your License*
- *Create a Repository*
- *Load Your Data*
 - *Load data through the GraphDB Workbench*
 - *Load data through SPARQL or RDF4J API*
 - *Load data through the GraphDB LoadRDF tool*
- *Explore Your Data and Class Relationships*
 - *Explore instances*
 - *Create your own visual graph*
 - *Class hierarchy*
 - *Domain-Range graph*
 - *Class relationships*
- *Query Your Data*
 - *Query data through the Workbench*
 - *Query data programmatically*
- *Additional Resources*

2.1 Run GraphDB as a Standalone Server

What's in this document?

- *Running GraphDB*
 - *Options*
- *Configuring GraphDB*

- *Paths and network settings*
- *Java virtual machine settings*
- *Stopping the database*

The default way of running GraphDB is as a standalone server. The server is platform-independent, and includes all recommended JVM parameters for immediate use.

2.1.1 Running GraphDB

1. Download your GraphDB distribution file and unzip it.
2. Start the GraphDB Server and Workbench interface by executing the graphdb startup script located in the \$graphdb_home/bin folder:

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

2.1.1.1 Options

The startup script supports the following options:

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

Note: Run `graphdb -s` to start GraphDB in server-only mode without the web interface (no workbench). A remote workbench can still be attached to the instance.

2.1.2 Configuring GraphDB

2.1.2.1 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:

```
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 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.

2.1.2.2 Java virtual machine settings

We strongly recommend to set 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 `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 tested GraphDB's performance against the LDBC benchmark with newer versions (i.e., JDK 11 or higher), observing a drop when using G1GC. Tests were also conducted with the Parallel Garbage Collector (ParallelGC), this time producing normal results, i.e., no performance drop.

This is why we recommend experimenting with garbage collectors if using JDK 11 or higher, so as to find the option that would provide you with an optimal configuration.

2.1.3 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.

2.2 Set up Your License

GraphDB SE is available under an RDBMS-like commercial license on a per-server-CPU basis. It is neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact GraphDB-info@ontotext.com.

When installing GraphDB SE, the license file can be set through the GraphDB Workbench or programmatically:

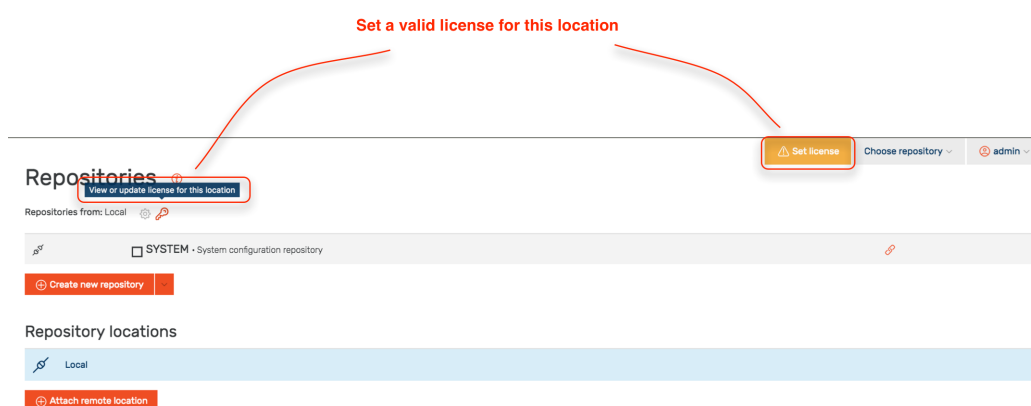
No license was set
Please supply a valid license or contact Ontotext AD sales department to acquire one.

[Set new license](#)

The software is provided "AS IS", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

To do that, follow the steps:

1. Add, view or update your license by clicking on the key icon next to the active location path.



2. Select the license file and register it.

Register GraphDB License ?

Click here to upload your license file
Supported file format is .license

or copy your license code below

[Register](#)
[Cancel](#)

You can also copy and paste it in the text area.

Register GraphDB License ?

 [Click here to upload your license file](#)
Supported file format is .license

or copy your license code below

```
cC/cSSw8Xae8wQt++3Ah756UAJX50YxHVyDLwKa+Jdjnl8ZMjqin+1CUmiuPSTfcq7Be4plrqe1H2IT
Hv9v3TVaU6SJEK+adsad73854adsas98456jklsdfEE+0CtJnV0e4y2pl2dukvUatNN91ws4KaQlra+E1p
3VKkm2ITrd6PtxwIS58I0HLIPPbbJ2SvSCLZUyg2Flx2/E2YV7vAUxMDrAtWKWINKzXrMJ1lun9MRDwL
uswoC0QxbUMtyUgZXYveo6cMloot30FU9in4d0sTmsidufosnGI/+EFAEt0+HiRouFN/h+PeXRyvirKS
2tuouQ6Lsm209EILpeTyjyM1zQ/xQ/B79rJGixMhHuCF970LR0VjWN2NnvajKSotUnmzn0tCwazz1f2k
LBfY2t/m305qQ4l6TpCKolGkQfGg5o6ha+f0p3G74UNqk03sgAcp7gKKXt9ntYvr6W4Ayo7i16H87j7e
SMgNyCq9645ERZfwqLS9iodwbRsgf0U6CcJ1ia2u0e4kki+b89YbxIT4F0libQXuMvIwdCibazuBM3d+
eqA01uawIYk0BD2SjPLmXyjM1hMaXjlr5XGfD/Hyhys=
```

Register

Cancel

3. Validate your license.

Are your license details correct? ×

Licensed to

**Ontotext Internal - allowed only for internal
development and testing**

Valid until

2017-12-31

Number of cores

Unlimited

Non-commercial research only

Cancel

Yes

4. After completing these steps, you will see the details of your license.

2.3 Create a Repository

Now let's create your first repository.

Hint: When started, GraphDB creates *GraphDB-HOME/data* directory as an active location. To change the directory, see *Configuring GraphDB Data Directory*.

1. Go to *Setup -> Repositories*.
2. Click *Create new repository*.

Repositories ?

Repositories from: Local  

  **SYSTEM** • System configuration repository 

 Create new repository

Create from file

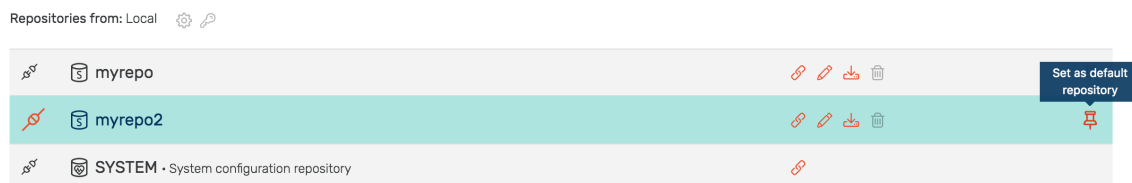
3. Enter **myrepo** as a *Repository ID* and leave all other optional configuration settings at their default values.

Tip: For repositories with over several tens of millions of statements, see [Configuring a Repository](#).

- Click the *Connect* button to set the newly created repository as the repository for this location.



- 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 Workbench REST API.

2.4 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> - by *Get RDF data from a URL* from *User data* tab of the *Import* page.

2.4.1 Load data through the GraphDB Workbench

Load data from local files

Let's load your data.

- Go to *Import* -> *RDF*.
- Open the *User data* tab and click the *Upload RDF files* to upload the files from the **News** sample dataset provided in the distribution folder.

Import ?

User data

Server files

Help

Upload RDF files
 All RDF formats, up to 200 MB

Get RDF data from a URL
 All RDF formats

Import RDF text snippet
 Type or paste RDF data

☐

Import

Reset status

Remove

☒	graphdb-news-dataset.nt Imported successfully in less than a second.	Import
☒	pub-ontology-types.ttl Imported successfully in less than a second.	Import
☒	pub-ontology.ttl Imported successfully in less than a second.	Import
☒	pub-properties.ttl Imported successfully in less than a second.	Import
☒	publishing-ontology.ttl Imported successfully in less than a second.	Import

- Click the *Import* button.
- Enter the *Import settings* in the pop-up window.

Import settings ×

Base IRI ?

http://exampleuri.com/examplepath

Target graph ?

☒ From data
 ☐ The default graph
 ☐ Named graph

http://example.com/graph...

☐ Enable replacement of existing data

Show advanced settings ∨

Restore defaults

Cancel

Import

Import Settings

- Base URI:** 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](#).

- Start importing by clicking the *Import* button.

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.
-

2.4.2 Load data through SPARQL or RDF4J API

The GraphDB database also supports a very 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:
 - select *Setup -> Repositories*
 - click the link icon next to the repository name



- copy the repository URL.
2. Go to the folder where your local data files are.
 3. Execute the script:

```
curl -X POST -H "Content-Type:application/x-turtle" -T <localfilename.ttl>  
http://localhost:7200/repositories/repository-id/statements
```

where `localfilename.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.

2.4.3 Load data through the GraphDB LoadRDF tool

LoadRDF 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 the [Loading data using the LoadRDF tool](#).

Note: Loading data through the GraphDB LoadRDF tool can be performed only if the repository is empty, e.g., the initial loading after the database has been inactive.

2.5 Explore Your Data and Class Relationships

2.5.1 Explore instances

To explore instances and their relationships, 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.



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 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

20

Preferred languages

en Add a language tag

☒ Show Predicate Labels

Types

Predicates

Preferred types

http://dbpedia.org/ontology/Person
Add preferred type

☒ Show Preferred Types Only

Ignored types

Add ignored type

Reset

Save

2.5.2 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 -> Create graph config*. Use the sample queries to guide you in the configuration.

Create visual graph config

Config name

My graph config (required)

Starting point

Graph expansion

Node basics

Edge basics

Node extra



http://...

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.

Save

Next 

Cancel

The following parts of the graph can be configured:

- Starting point - this is the initial state of your graph.
 - Search box - start with a search box to choose a different start resource each time;
 - Fixed node - you may want to start exploration with the same resource each time;
 - Query results - the initial config state may be the visual representation of a Graph SPARQL query result.
- Graph expansion - determines how new nodes and links are added to the visual graph when the user expands an existing node. The `?node` variable is required and will be replaced with the IRI of the expanded node.
- Node basics - this `SELECT` query controls how the type, label, comment and rank are obtained for the nodes in the graph. Node types correspond to different colours. 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.

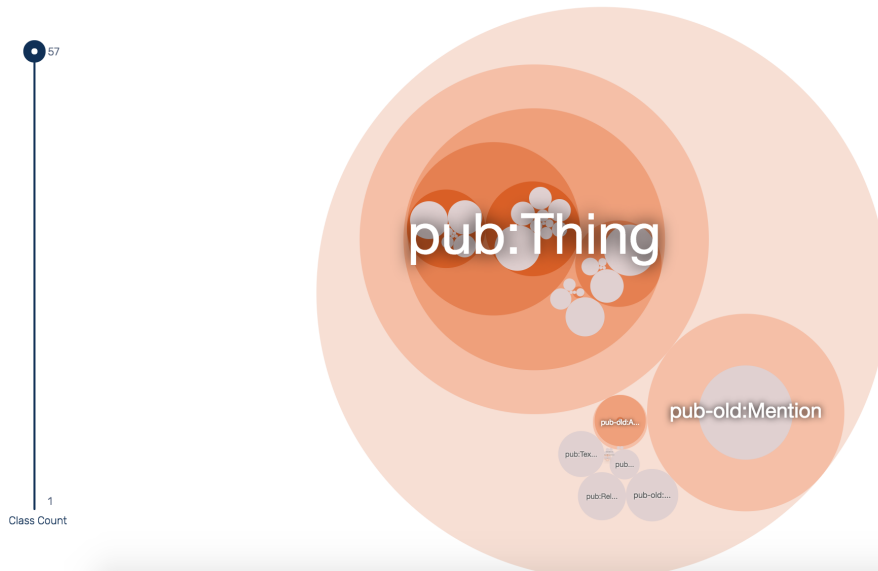
2.5.3 Class hierarchy

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, URI, and a list of its first 1000 class instances. The class instances are represented by their URIs, which, when clicked, lead to another view where you can further explore their metadata.

Class hierarchy ⓘ

🔍 🔄 🔍 🔍 🔍



The side panel includes the following:

- Local name;
- URI (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 1000 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 URI from the side panel.

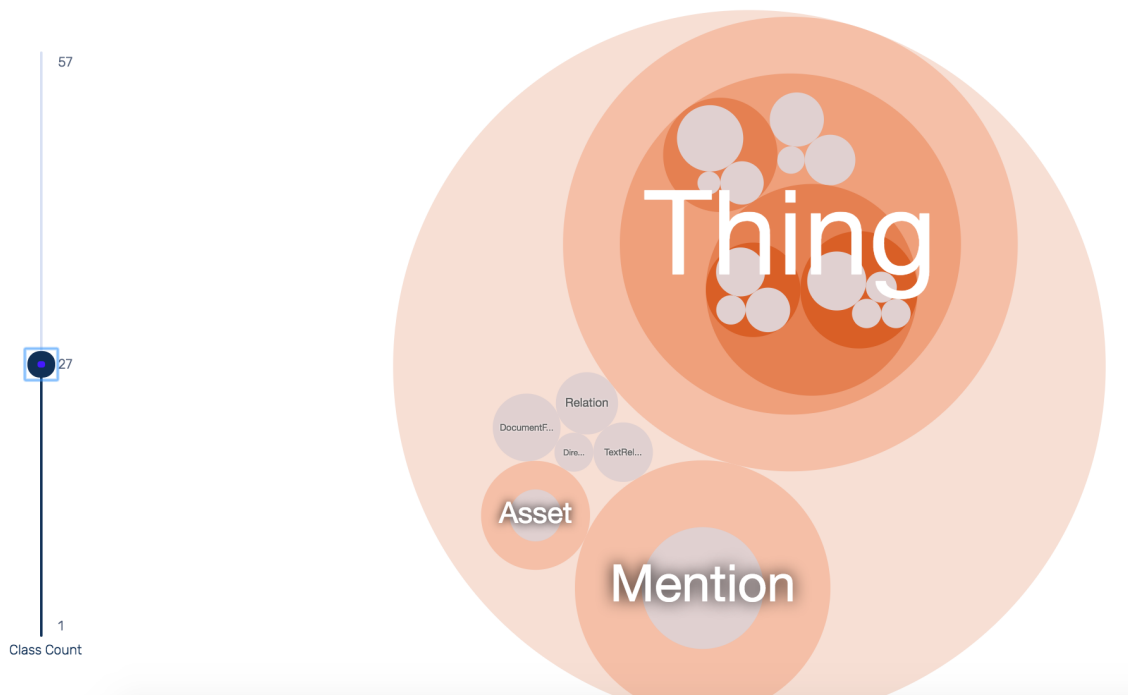
Facebook

Source: <http://ontology.ontotext.com/resource/tsk9gvw0q4n4>

subject	predicate	object	context	all
1	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	pub:coordinateLocation	http://ontology.ontotext.com/resource/Q355S32C46011-FD59-441C-9E0A-B8F5E22A0C42	http://www.ontotext.com/explicit
2	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	pub:logoImage	http://ontology.ontotext.com/resource/Q355SDAD9B293-1DD6-42A7-AC9B-E8F190CD9FEC	http://www.ontotext.com/explicit
3	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	pub:preferredLabel	Facebook	http://www.ontotext.com/explicit
4	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	pub:website	http://ontology.ontotext.com/resource/Q355S3a882975-4752-9eda-c2b0-e7840d0c2865	http://www.ontotext.com/explicit
5	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	pub:Agent	http://www.ontotext.com/explicit
6	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	pub:Company	http://www.ontotext.com/explicit
7	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	pub:Concept	http://www.ontotext.com/explicit
8	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	pub:Organization	http://www.ontotext.com/explicit
9	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	pub:Thing	http://www.ontotext.com/explicit
10	http://ontology.ontotext.com/resource/tsk9gvw0q4n4	rdf:type	owl:Thing	http://www.ontotext.com/explicit

- 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.

Class hierarchy



- To administer your data view, use the toolbar options on the right-hand side of the screen.

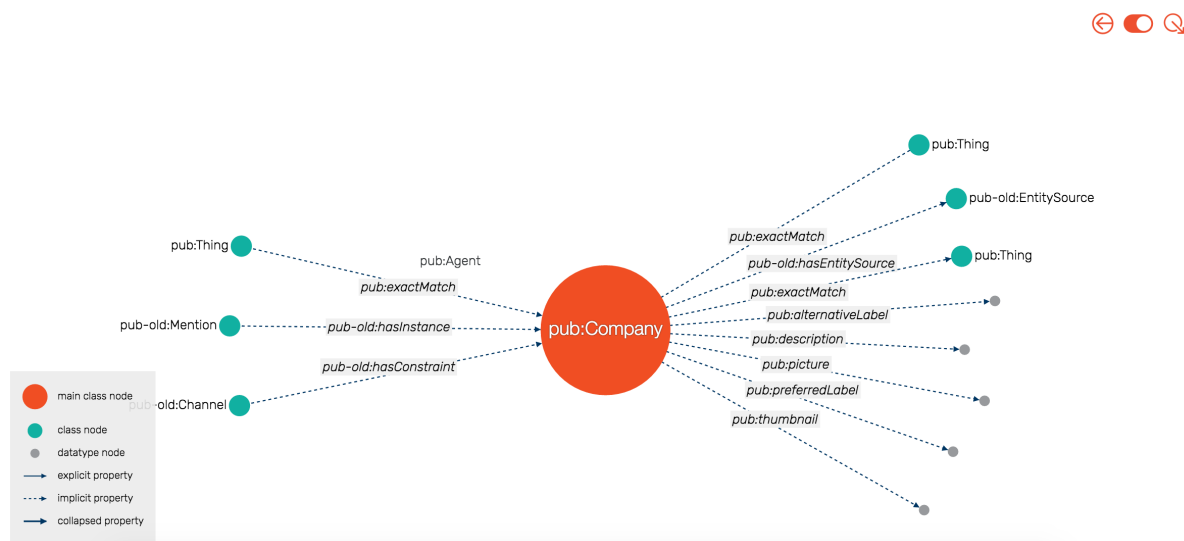


- To see only the class labels, click the *Hide/Show Prefixes*. You can still view the prefixes when you hover over the class that interests you.
- To zoom out of a particular class, click the *Focus diagram* home icon.
- To reload the data on the diagram, click the *Reload diagram* icon. This is recommended 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.

2.5.4 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>`.

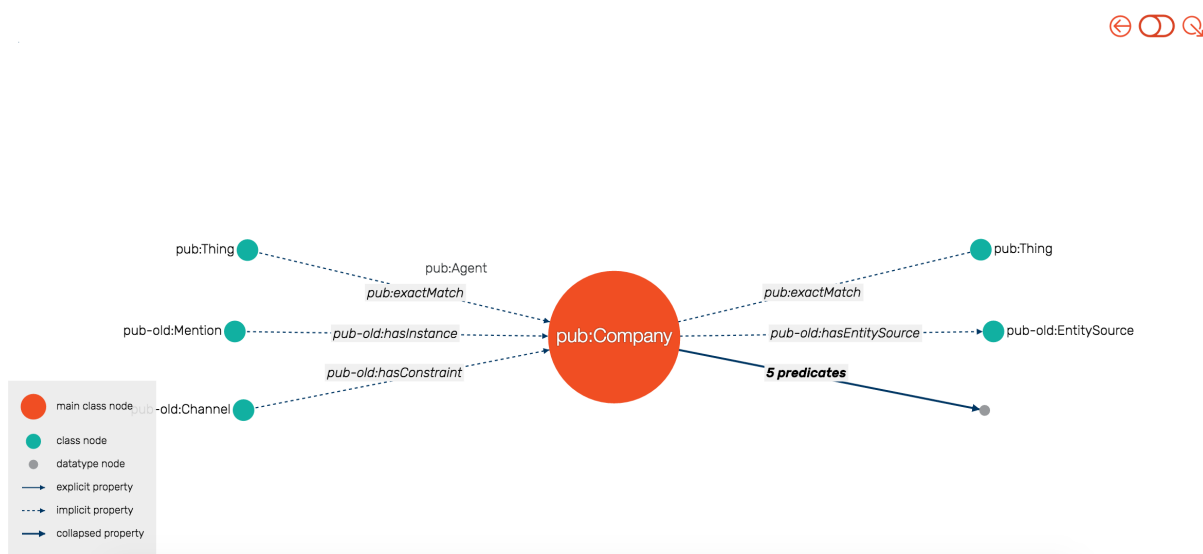
Domain-Range graph ⓘ



You can also further explore the class connectedness by clicking:

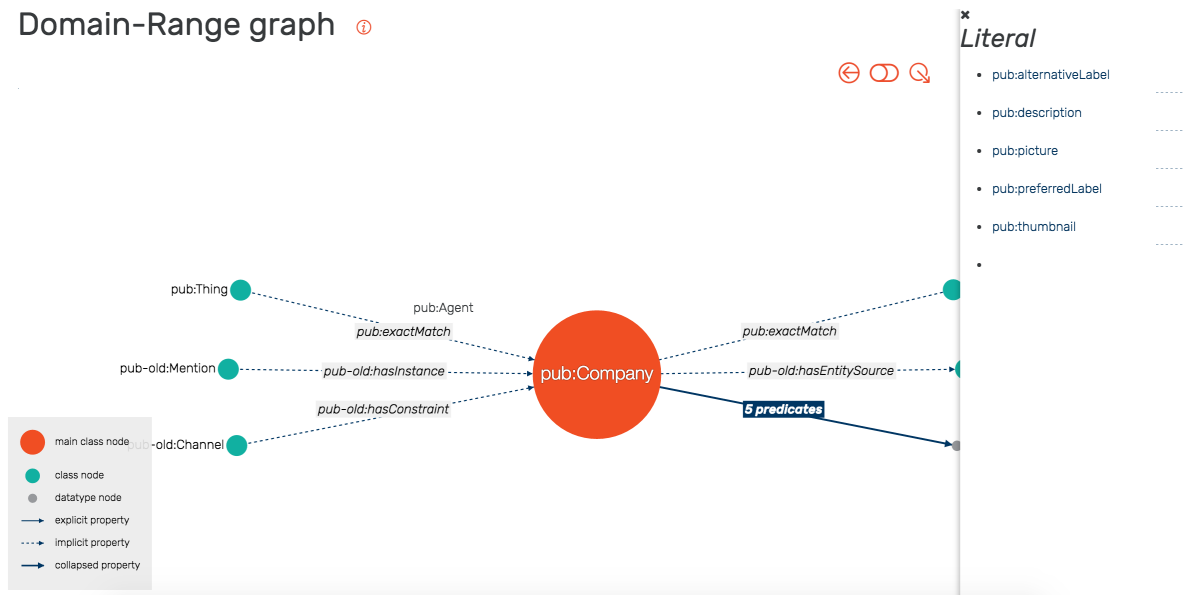
- the green nodes (*object property class*);
- the labels - they lead to the *View resource* page, where you can find more information about the current class or property;
- the slider *Show collapsed predicates* to hide all edges sharing the same source and target nodes;

Domain-Range graph ⓘ



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.

Domain-Range graph ⓘ



Administering the diagram view

To administer your diagram view, use the toolbar options on the right-hand side of the screen.

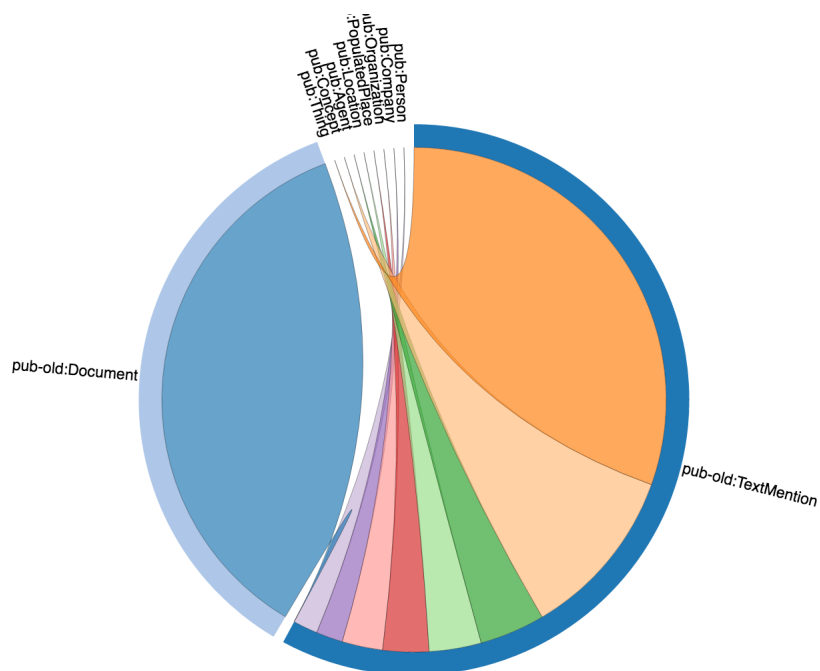


- To go back to your class in the Class hierarchy, click the *Back to Class hierarchy diagram* button.
- To collapse edges with common source/target nodes, in order to see the diagram more clearly, click the *Show all predicates/Show collapsed predicates* button. The default is collapsed.
- To export the diagram as an .svg image, click the *Export Diagram* download icon.

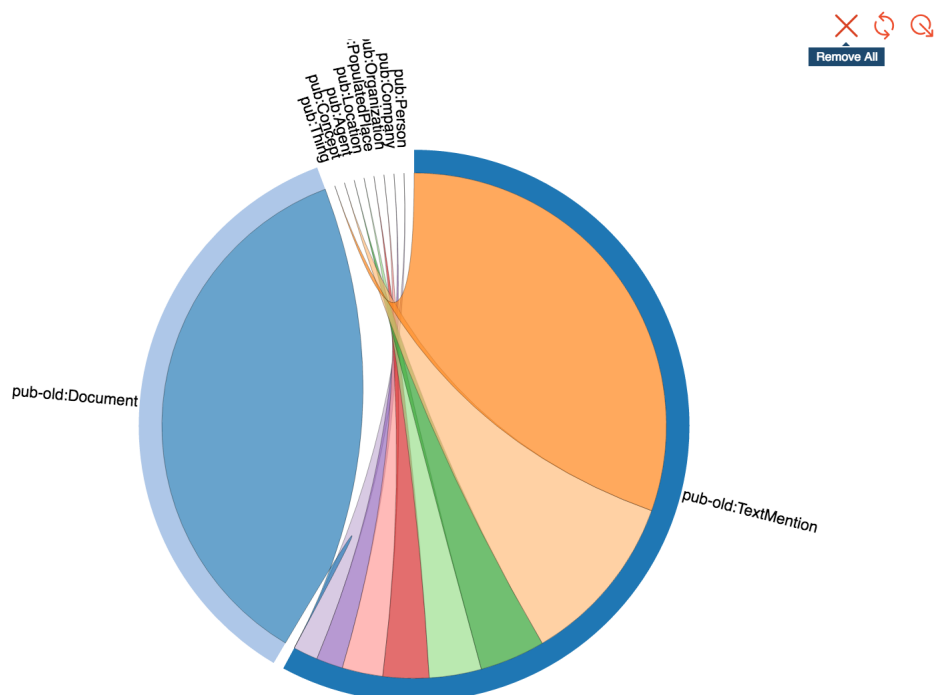
2.5.5 Class relationships

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 colour 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.

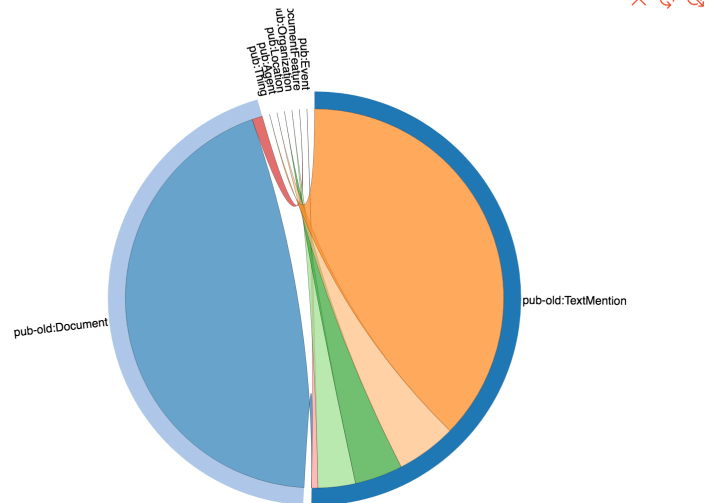
Class relationships ⓘ

Showing the dependencies between 8 classes

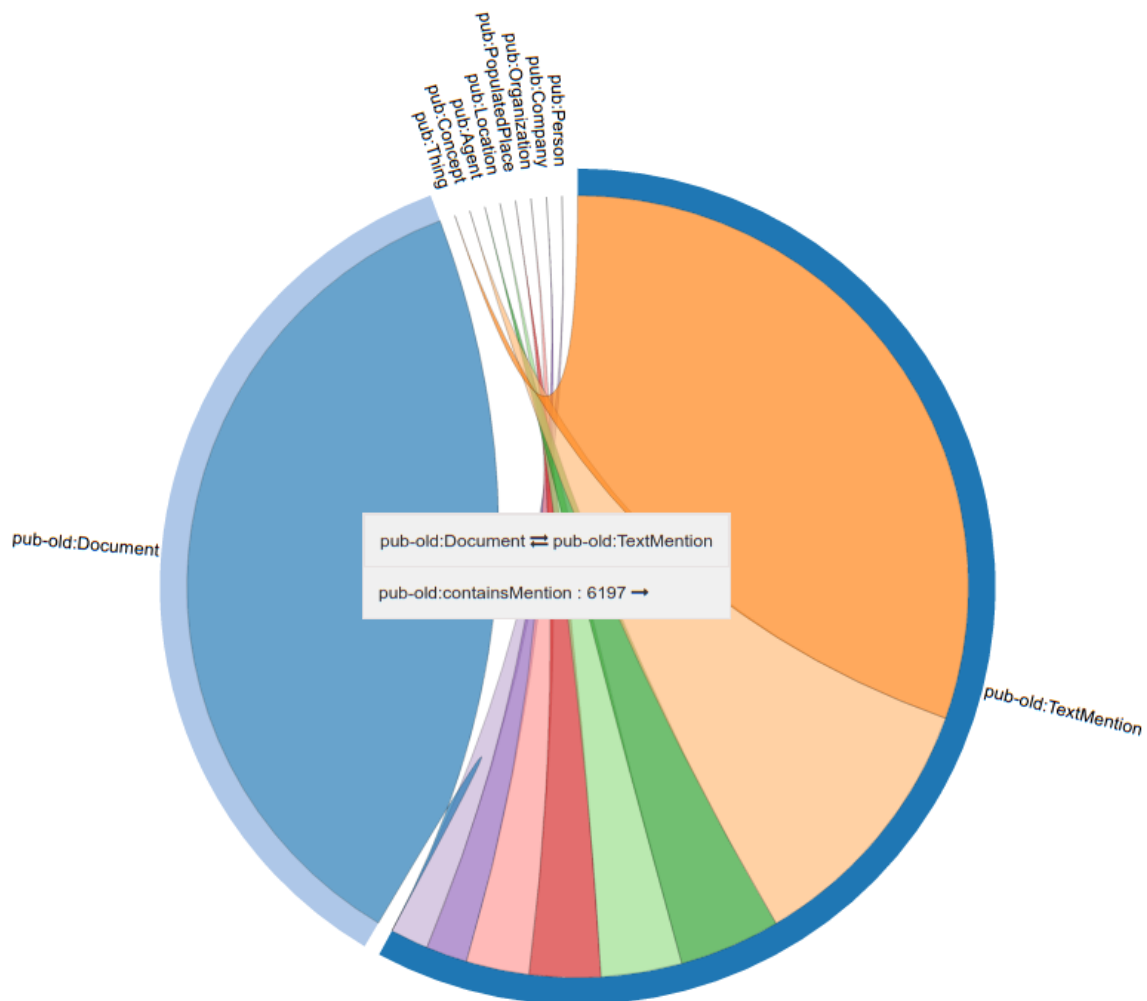
Filter classes

☒ All
 ☐ Incoming
 ☐ Outgoing

Class	Links	
pub-old:TextMention	17K	↔
pub-old:Document	6K	→
pub:Thing	5K	→
pub:Concept	2K	→
pub:Agent	762	→
pub:Location	592	→
pub:PopulatedPlace	523	→
pub:Organization	464	→
pub:Company	302	→
pub:Person	298	→
pub:Relation	289	↔
pub:Work	135	→
pub-old:DocumentFeature	126	→
pub:Athlete	81	→
pub:Event	76	→

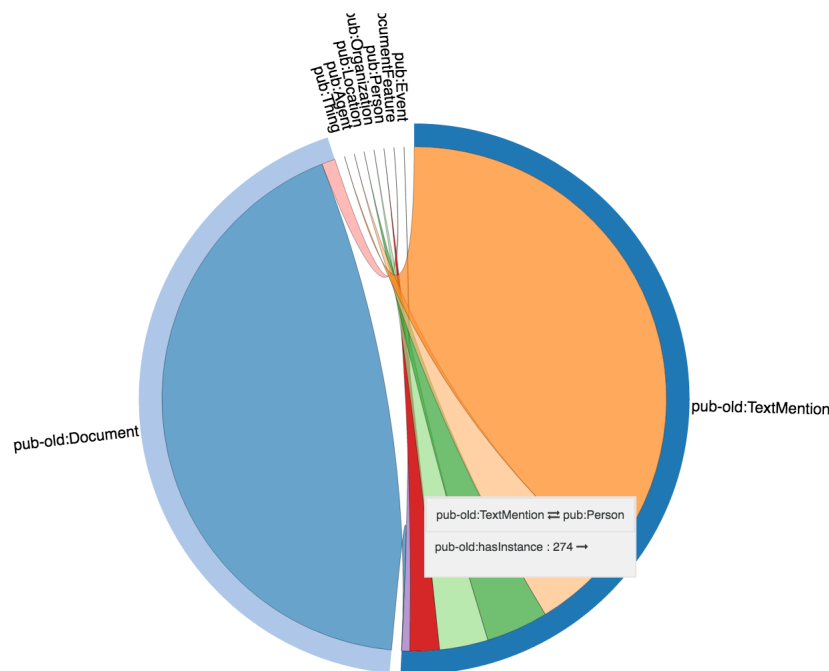


To see how many annotations (mentions) there are in the documents, click on the blue bundle representing the relationship between the classes **pub-old:Document** and **pub-old:TextMention**. The tooltip shows that there are 6,197 annotations linked by the **pub-old:containsMention** predicate.



To see how many of these annotations are about people, click on light purple bundle representing the relationship between the classes **pub-old:TextMention** and **pub:Person**. The tooltip shows that 274 annotations are about

people linked by the **pub-old:hasInstance** predicate.



2.6 Query Your Data

2.6.1 Query data through the Workbench

Hint: SPARQL is an 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's time to delve into your data. The following is one possible scenario for querying it.

1. Select the repository you want to work with, in this example **News**, and click the *SPARQL* menu tab.
2. Let's say you are interested in people. Paste the query below into the query field, and click Run to find all people mentioned in the documents from this news articles dataset.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?x ?Person where {
  ?x a pub:Person .
  ?x pub:preferredLabel ?Person .
  ?doc pub-old:containsMention / pub-old:hasInstance ?x .
}
```

	x	Person
1	http://ontology.ontotext.com/resource/tsk9hdnas934	"Fernando Alonso"@en
2	http://ontology.ontotext.com/resource/tsk9gxv1ughs	"Ana Bohueles"@en
3	http://ontology.ontotext.com/resource/tsk8hh9y46io	"Romain Grosjean"@en
4	http://ontology.ontotext.com/resource/tsk8ohbx7sw	"Sebastian Vettel"@en
5	http://ontology.ontotext.com/resource/tsm846z3gav4	"Jules Bianchi"@en
6	http://ontology.ontotext.com/resource/tsk5db9xmo0	"Jenson Button"@en
7	http://ontology.ontotext.com/resource/tsk4vi4csp34	"Ben Bernanke"@en
8	http://ontology.ontotext.com/resource/tsmkyhyokglc	"Maurice R. Greenberg"@en
9	http://ontology.ontotext.com/resource/tsm5f2nv45xc	"Jeff Bezos"@en
10	http://ontology.ontotext.com/resource/tsk4v59zod1c	"Amedeo Modigliani"@en
11	http://ontology.ontotext.com/resource/tskvz205mu4g	"Alberto Giacometti"@en
12	http://ontology.ontotext.com/resource/tsk6uzzsjn4	"Larry King"@en
13	http://ontology.ontotext.com/resource/tsk7tp27ytc	"Mark Rothko"@en
14	http://ontology.ontotext.com/resource/tsmf5me6cydc	"Jeff Koons"@en
15	http://ontology.ontotext.com/resource/tsk51g6mifwg	"Vincent Willem Van Gogh"@en
16	http://ontology.ontotext.com/resource/tslz2d6btvk	"Damien Hirst"@en
17	http://ontology.ontotext.com/resource/tsk4vd64awhs	"Andy Warhol"@en
18	http://ontology.ontotext.com/resource/tsk7xn0p0jr4	"Pablo Picasso"@en
19	http://ontology.ontotext.com/resource/tsk8jo76w9hc	"Salvador Dalí"@en
20	http://ontology.ontotext.com/resource/tsm5chweb01s	"Jean Dubuffet"@en
21	http://ontology.ontotext.com/resource/tsk78dfdet4w	"Marlon Brando"@en

- 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. }
```

- Find all people mentioned in the documents, ordered by popularity in the repository.

```
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
select distinct ?x ?PersonLabel ?rank where {
  ?x a pub:Person .
  ?x pub:preferredLabel ?PersonLabel .
  ?doc pub-old:containsMention / pub-old:hasInstance ?x .
  ?x rank:hasRDFRank ?rank .
} ORDER by DESC (?rank)
```

	x	PersonLabel	rank
1	http://ontology.ontotext.com/resource/tsk4u7qfztvk	"Abdullah Öcalan (orospu cocugu)"@en	"0.00" xsd:float
2	http://ontology.ontotext.com/resource/tskriiug3kmo	"Adrian Peterson"@en	"0.00" xsd:float
3	http://ontology.ontotext.com/resource/tskvz205mu4g	"Alberto Giacometti"@en	"0.00" xsd:float
4	http://ontology.ontotext.com/resource/tsk4v3ayilio0	"Alfred Nobel"@en	"0.00" xsd:float
5	http://ontology.ontotext.com/resource/tsk4v59zod1c	"Amedeo Modigliani"@en	"0.00" xsd:float
6	http://ontology.ontotext.com/resource/tsk9gxv1ughs	"Ana Bohueles"@en	"0.00" xsd:float
7	http://ontology.ontotext.com/resource/tsk4vd64aups	"Andy Murray"@en	"0.00" xsd:float
8	http://ontology.ontotext.com/resource/tsk4vd64awhs	"Andy Warhol"@en	"0.00" xsd:float
9	http://ontology.ontotext.com/resource/tsk4vb735ibk	"Angela Merkel"@en	"0.00" xsd:float
10	http://ontology.ontotext.com/resource/tsk4vd64aosg	"Angelina Jolie"@en	"0.00" xsd:float
11	http://ontology.ontotext.com/resource/tsk4wyetflog	"Barack Obama"@en	"0.00" xsd:float
12	http://ontology.ontotext.com/resource/tsk4xa887cw0	"Bashar al-Assad"@en	"0.00" xsd:float
13	http://ontology.ontotext.com/resource/tsk4xi4csp34	"Ben Bernanke"@en	"0.00" xsd:float
14	http://ontology.ontotext.com/resource/tsldjpb6b5s	"Bertrand de Billy"@en	"0.00" xsd:float
15	http://ontology.ontotext.com/resource/tslg78agai8g	"Brian Wansink"@en	"0.00" xsd:float
16	http://ontology.ontotext.com/resource/tslhkzyrdzwg	"CY Leung"@en	"0.00" xsd:float
17	http://ontology.ontotext.com/resource/tsliffy7voj	"Carrie Lam"@en	"0.00" xsd:float
18	http://ontology.ontotext.com/resource/tsltz2d6btvk	"Damien Hirst"@en	"0.00" xsd:float
19	http://ontology.ontotext.com/resource/tsk59ea5gqo0	"Daniel Ricciardo"@en	"0.00" xsd:float
20	http://ontology.ontotext.com/resource/tslm2gp4xybk	"Danil Kvart"@en	"0.00" xsd:float
21	http://ontology.ontotext.com/resource/tsk594ezu5lc	"David Ferrer"@en	"0.00" xsd:float

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

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

```

?person pub:preferredLabel ?personLabel .
?person pub:memberOfPoliticalParty ?party .
?party pub:hasValue ?value .
?value pub:preferredLabel ?partyLabel .
}

```

	personLabel	partyLabel
1	"Manuel Valls"@en	"Socialist Party"@en
2	"Sylvia Mathews Burwell"@en	"Democratic Party"@en
3	"Shinzō Abe"@en	"Liberal Democratic Party"@en
4	"Xi Jinping"@en	"Communist Party of China"@en
5	"François Hollande"@en	"Socialist Party"@en
6	"Li Keqiang"@en	"Communist Party of China"@en
7	"Abdullah Öcalan (orospu cocugu)"@en	"Kurdistan Workers' Party"@en
8	"Ben Bernanke"@en	"Republican Party"@en
9	"Eli Lilly"@en	"Republican Party"@en
10	"Marlon Brando"@en	"Democratic Party"@en
11	"Recep Tayyip Erdoğan"@en	"Justice and Development Party"@en
12	"Bashar al-Assad"@en	"Ba'ath Party"@en
13	"Judy Chu"@en	"Democratic Party"@en
14	"Angela Merkel"@en	"Democratic Awakening"@en
15	"Angela Merkel"@en	"Christian Democratic Union"@en
16	"William Hague"@en	"Conservative Party"@en
17	"Jeh Johnson"@en	"Democratic Party"@en
18	"Jim Inhofe"@en	"Republican Party"@en
19	"Barack Obama"@en	"Democratic Party"@en
20	"Tom Frieden"@en	"Democratic Party"@en

6. Did you know that Marlon Brando was from the Democratic Party? Find what other mentions occur together with Marlon Brando in the given news article.

```

PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?Mentions where {
<http://www.reuters.com/article/2014/10/06/us-art-auction-idUSKCN0HV21B20141006> pub-
-old:containsMention / pub-old:hasInstance ?x .
?x pub:preferredLabel ?Mentions .
}

```

	Mentions
1	"New York City"@en
2	"Andy Warhol"@en
3	"Pablo Picasso"@en
4	"Salvador Dalí"@en
5	"Eastern Time Zone"@en
6	"Reuters"@en
7	"Jeff Koons"@en
8	"Jean Dubuffet"@en
9	"France"@en
10	"Sotheby's"@en
11	"Museum of Fine Arts"@en
12	"Alberto Giacometti"@en
13	"Mark Rothko"@en
14	"Palace of Versailles"@en
15	"Hong Kong"@en
16	"European Union"@en
17	"Europe"@en
18	"Marlon Brando"@en
19	"Schlumberger"@en
20	"Museum of Modern Art"@en
21	"world"@en

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

```

PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
select distinct ?p ?objectLabel where {
<http://ontology.ontotext.com/resource/tsk78dfdet4w> ?p ?o .
}

```

```
{
  ?o pub:hasValue ?value .
  ?value pub:preferredLabel ?objectLabel .
  } union {
    ?o pub:hasValue ?objectLabel .
    filter (isLiteral(?objectLabel)) .
  }
}
```

	p	objectLabel
1	pub-genericProperty	"Tarita Terlipia"@en
2	pub-genericProperty	"Actors Studio"@en
3	pub-genericProperty	"The New School"@en
4	pub-genericProperty	"Jocelyn Brando"@en
5	pub-genericProperty	"Shattuck-Saint Mary's"@en
6	pub-genericProperty	"Marlon Brando, Sr."@en
7	pub-genericProperty	"Omaha"@en
8	pub-genericProperty	"Democratic Party"@en
9	pub-genericProperty	"Los Angeles"@en
10	pub-genericProperty	"United States of America"@en
11	pub-genericProperty	"actor"@en
12	pub-genericProperty	"film director"@en
13	pub-genericProperty	"Movita Castaneda"@en
14	pub-genericProperty	"Anna Kashfi"@en
15	pub-placeOfBirth	"Omaha"@en
16	pub-placeOfDeath	"Los Angeles"@en
17	pub-countryOfCitizenship	"United States of America"@en
18	pub-occupation	"actor"@en
19	pub-occupation	"film director"@en
20	pub-almaMater	"Actors Studio"@en
21	pub-almaMater	"The New School"@en

8. Find all documents that mention members of the Democratic Party and the names of these people.

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

	document	personLabel
1	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Sylvia Mathews Burwell"@en
2	http://www.reuters.com/article/2014/10/06/us-art-auction-idUSKCN0HV21B20141006	"Marlon Brando"@en
3	http://www.reuters.com/article/2014/10/10/us-usa-california-mountains-idUSKCN0HZ0U720141010	"Judy Chu"@en
4	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Jeh Johnson"@en
5	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Barack Obama"@en
6	http://www.reuters.com/article/2014/10/10/us-usa-california-mountains-idUSKCN0HZ0U720141010	"Barack Obama"@en
7	http://www.reuters.com/article/2014/10/10/us-health-ebola-usa-idUSKCN0HY2A520141010	"Tom Frieden"@en

9. Find when these people were born and died.

```
PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
select distinct ?person ?personLabel ?dateOfBirth ?dateOfDeath where {
  ?document pub-old:containsMention / pub-old:hasInstance ?person .
  ?person pub:preferredLabel ?personLabel .
  OPTIONAL {
    ?person pub:dateOfBirth / pub:hasValue ?dateOfBirth .
  }
  OPTIONAL {
    ?person pub:dateOfDeath / pub:hasValue ?dateOfDeath .
  }
}
```

```

?person pub:memberOfPoliticalParty / pub:hasValue / pub:preferredLabel "Democratic Party
↪"@en .
} order by ?dateOfBirth

```

	person	personLabel	dateOfBirth	dateOfDeath
1	http://ontology.ontotext.com/resource/tsk78dfde4w	"Marlon Brando"@en	"1924-04-03" xs:date	"2004-07-01" xs:date
2	http://ontology.ontotext.com/resource/tsm835hi3e3k	"Judy Chu"@en	"1953-07-07" xs:date	
3	http://ontology.ontotext.com/resource/tsm5h3o0r9xc	"Jeh Johnson"@en	"1957-09-11" xs:date	
4	http://ontology.ontotext.com/resource/tsk4wyefitog	"Barack Obama"@en	"1961-08-04" xs:date	
5	http://ontology.ontotext.com/resource/tsnhtzmzu9uyo	"Sylvia Mathews Burwell"@en	"1965-06-01" xs:date	
6	http://ontology.ontotext.com/resource/tsn6fvmpp4ow	"Tom Frieden"@en	"1960" xs:gYear	

Tip: You can play with more example queries from the `Example_queries.rtf` file provided in the distribution folder.

2.6.2 Query data programmatically

SPARQL is not only a standard query language, but also a protocol for communicating with RDF databases. GraphDB stays compliant with the protocol specification, and allows querying data with standard HTTP requests.

Execute the example query with an HTTP GET request:

```

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

```

Execute the example query with a POST operation:

```

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

```

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.

2.7 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>

INSTALLATION

3.1 Requirements

What's in this document?

- *Minimum requirements*
- *Hardware sizing*
- *Licensing*

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 8, 11, or 12 (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:

- Statements are the planned number of explicit statements.
- Unique resources are the expected number of unique RDF resources (IRIs, blank nodes, literals).
- Java heap (minimal) is the minimal recommend JVM heap required to operate the database controlled by `-Xmx` parameter.
- Java heap (optimal) is the recommended JVM heap required to operate a database controlled by `-Xmx` parameter.
- Off heap is the database memory footprint (outside of the JVM heap) required to initialize the database.
- OS is the recommended minimal space reserved for the operating system.

- Total is the RAM required for the hardware configuration.
- Repository image is the expected size on disk. For repositories with inference use the total number of explicit + implicit statements.

State-ments	Unique resources	Java heap (min)	Java heap (opt)	Off heap	OS	Total	Repository image
100M	33.3M	1.2GB	3.6GB	370M	2	6GB	12GB
200M	66.6M	2.4GB	7.2GB	740M	3	11GB	24GB
500M	166.5M	6GB	18GB	1.86GB	4	24GB	60GB
1B	333M	12GB	30GB	3.72GB	4	38GB	120GB
2B	666M	24GB	30GB	7.44GB	4	42GB	240GB
5B	1.665B	30GB	30GB	18.61GB	4	53GB	600GB
10B	3.330B	30GB	30GB	37.22GB	4	72GB	1200GB
20B	6.660B	30GB	30GB	74.43GB	4	109GB	2400GB

3.1.3 Licensing

GraphDB SE is available under an RDBMS-like commercial license on a per-server-CPU basis. It is neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact [GraphDB-info@ontotext.com](mailto:info@ontotext.com).

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 standalone server - For production use, we recommend installing the standalone server. The installation comes with a preconfigured web server. This is the standard way to use GraphDB if you plan to use the database for longer periods with preconfigured log files.

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

3.2.1 Run GraphDB as a Standalone Server

What's in this document?

- *Running GraphDB*
 - *Options*
- *Configuring GraphDB*
 - *Paths and network settings*
 - *Java virtual machine settings*
- *Stopping the database*

The default way of running GraphDB is as a standalone server. The server is platform-independent, and includes all recommended JVM parameters for immediate use.

3.2.1.1 Running GraphDB

1. Download your GraphDB distribution file and unzip it.

2. Start the GraphDB Server and Workbench interface by executing the graphdb startup script located in the \$graphdb_home/bin folder:

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

Options

The startup script supports the following options:

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

Note: Run `graphdb -s` to start GraphDB in server-only mode without the web interface (no workbench). A remote workbench can still be attached to the instance.

3.2.1.2 Configuring GraphDB

Paths and network settings

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

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

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

Note: The order of precedence for GraphDB configuration properties is as follows: command line supplied arguments > `GDB_JAVA_OPTS` > config file.

The GraphDB home directory

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

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

- Running as a standalone server: the default is the same as the distribution directory.
- All other types of installations: OS-dependent directory.
 - On Mac: `~/Library/Application Support/GraphDB`.

- On Windows: \Users\<username>\AppData\Roaming\GraphDB.
- On Linux and other Unixes: ~/.graphdb.

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

Java virtual machine settings

We strongly recommend to set 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 `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 tested GraphDB's performance against the LDBC benchmark with newer versions (i.e., JDK 11 or higher), observing a drop when using G1GC. Tests were also conducted with the Parallel Garbage Collector (ParallelGC), this time producing normal results, i.e., no performance drop.

This is why we recommend experimenting with garbage collectors if using JDK 11 or higher, so as to find the option that would provide you with an optimal configuration.

3.2.1.3 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.

3.3 Configuring GraphDB

GraphDB 8.x relies on several main directories for configuration, logging and data.

What's in this document?

- *Directories*
 - *GraphDB Home*
 - *Checking the configured directories*
- *Configuration*
 - *Config properties*
 - *Configuring logging*

- *Best practices*
 - *Step by step guide*

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`.
 - 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 subdirectory `data` 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 subdirectory `conf` 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 subdirectory `work` 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 subdirectory `logs` 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.

3.3.1.2 Checking the configured directories

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

```
GraphDB Home directory: /opt/test/graphdb-se-8.x.x
GraphDB Config directory: /opt/test/graphdb-se-8.x.x/conf
GraphDB Data directory: /opt/test/graphdb-se-8.x.x/data
GraphDB Work directory: /opt/test/graphdb-se-8.x.x/work
GraphDB Logs directory: /opt/test/graphdb-se-8.x.x/logs
```

3.3.2 Configuration

There is a single config file for GraphDB. GraphDB loads the config file `graphdb.properties` from the GraphDB config directory.

A sample file is provided in the distribution under `conf/graphdb.properties`.

3.3.2.1 Config properties

Config properties are defined in the config file 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 on the command line.

Note: The legacy properties (e.g. `owlim-license`) in the config file are ignored, but they work if specified as system properties.

List of configuration properties

General properties

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

graphdb.home defines the GraphDB home directory

graphdb.home.data defines the GraphDB data directory

graphdb.home.conf (only as a system property) defines the GraphDB conf directory

graphdb.home.work defines the GraphDB work directory

graphdb.home.logs defines the GraphDB logs directory

graphdb.workbench.home the place where the source for [GraphDB Workbench](#) is located

graphdb.license.file sets a custom path to the license file to use

graphdb.page.cache.size the amount of memory to be taken by the page cache

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.

graphdb.engine.entity-pool-implementation Defines the Entity Pool implementation for the whole installation. Possible values are “transactional” or “classic”. The default value is “transactional”. The “transactional-simple” implementation is not supported anymore.

graphdb.persistent.parallel.inferencers Since GraphDB 8.6.1, inferencers for our Parallel loader are shut down at the end of each transaction to minimize GraphDB's memory footprint. For cases where a lot of small insertions are done in a quick succession that can be a problem, as inferencer initialization times can be fairly slow. This setting reverts to the old behaviour where inferencers are only shut down when the repository is released. The default value is false.

graphdb.engine.entity.validate A global setting that ensures IRI validation in the entity pool. It is performed only when an IRI is seen for the first time (i.e., when being created in the entity pool). For consistency reasons, not only IRIs coming from RDF serializations, but also all new IRIs (via API or SPARQL), will be validated in the same way. This property can be turned off by setting its value to false.

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 config directory.

3.3.3 Best practices

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.

3.3.3.1 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` or set the `-D` option in Tomcat.

GraphDB creates the missing subdirectories `data`, `conf` (if you skipped that step), `logs`, and `work`.

3.4 Migrating GraphDB Configurations

If you want to migrate your GraphDB configurations and replicate the setup, there are three steps that you need to follow:

1. Back up your repository by copying the binary image (explained in more detail [here](#)), or restore it from an RDF export or from a binary image or zip backup (explained in more detail [here](#)).

For steps 2 and 3, it is useful to know that the GraphDB distribution package consists of several folders that are described in detail [here](#).

2. Copy the `config` directory to the new instance. It contains the `logback` and the GraphDB configuration.
3. Copy the `work` directory to the new instance. It contains all Workbench-related details, e.g., saved queries, users, user roles, etc.

Hint: To migrate from one GraphDB version to another, follow the steps in the last column of the table below.

Hint: When upgrading to a newer GraphDB version, it might contain plugins or connectors that are not present in the older version. In this case, the `PluginManager` 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> "pluginName" }"`.

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.

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

GraphDB	Connectors	Elastic-search	Lucene	Solr	Steps for migrating to a newer version
9.0.x	2.5.2	11.0.0	7.3.1	8.2.0	8.2.0 Recreate all the connectors with the <i>repair</i> option.
8.11.x	2.5.2	11.0.0	6.6.x	7.7.x	7.7.x
8.10.x	2.5.2	11.0.0	6.6.x	7.7.x	7.7.x
8.9.x	2.4.6	10.1.0	6.6.x	7.7.x	7.7.x Recreate all the connectors with the <i>repair</i> option.
8.8.x	2.4.2	10.0.0	6.3.x	7.4.x	7.4.x No special attention needed.
8.7.x	2.3.2	9.0.0	6.3.x	7.4.x	7.4.x Recreate all the connectors with the <i>repair</i> option. Need to rebuild Semantic similarity search indexes.
8.6.x	2.3.2	8.0.0	6.2.x	7.2.x	7.2.x Need to rebuild GeoSPARQL index.
8.5.x	2.2.4	7.2.0	5.3.x	6.5.x	6.5.x No more system repository in future installations. During the upgrade procedure, system repository will be removed while the data insight will be backed up. There will be <i>config.ttl</i> file in each repository directory with config data. Important points: 1. Cannot query System repo. 2. More strict parser - IRI validation according to RFC3987
8.4.x	2.2.2	7.2.0	5.3.x	6.5.x	6.5.x Autocomplete is no longer part of the fingerprint. In case of cluster, this will lead to out of sync for the workers. Stop the master and rename the <i>txlog</i> for backup purposes. Start the master again.
8.3.x	2.2.2	7.1.0	5.3.x	6.5.x	6.5.x No special attention needed.
8.2.x	2.2.1	7.0.0	5.3.x	6.5.x	6.5.x No special attention needed.
8.1.x	2.1.6	6.0.2	2.4.0	6.2.1	6.2.1 Recreate all the connectors with the <i>repair</i> option. Please note the breaking changes for each connector. https://www.elastic.co/guide/en/elasticsearch/reference/5.0/breaking-changes-5.0.html https://lucene.apache.org/solr/guide/6_6/upgrading-solr.html#upgrading-solr
8.0.x	2.0.3	6.0.2	2.4.0	6.2.1	6.2.1 No special attention needed. Important points: 1. The engine replaces PCSO and PCOS indexes with CPSO. 2. Remove the owlim-license parameter from the repository TTL configuration.
7.2.x		5.1.1	2.3.1	5.5.0	5.5.0 Ask for support
7.1.x		5.1.0	2.3.1	5.5.0	5.5.0 Ask for support
7.0.x		5.0.1	2.3.1	5.5.0	5.5.0 Ask for support
6.6.x		4.2.7	1.7.2	4.10.4	4.10.4 Ask for support
6.5.x		4.2.3	1.7.2	4.10.4	4.10.4 Ask for support
6.4.x		4.1.2	1.6.0	4.10.4	4.10.4 Ask for support
6.3.x		4.0.1	1.6.0	4.10.4	4.10.4 Ask for support
6.2.x		4.0.1			Ask for support
6.1.x		3.1.4			Ask for support
6.0.x		3.0.0		3.6.2	Ask for support

3.5 Distribution package

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

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

After the first successful database run, the following directories will be generated, unless their default value is not explicitly changed in `conf/graphdb.properties`.

Default path	Description
data/	Location of the repository data
logs/	Place to store of all database log files
work/	Work directory with non-user editable configurations

3.6 Using Maven Artifacts

What's in this document?

- *Public Maven repository*
- *Distribution*
- *GraphDB JAR file for embedding the database or plugin development*
- *I want to proxy the repository in my nexus*

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.

3.6.1 Public Maven repository

To browse and search the public GraphDB's Maven repository, use our [Nexus](#).

For the Gradle build script:

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

For the Maven POM file:

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

3.6.2 Distribution

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

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.10</version>
      <executions>
        <execution>
          <id>copy</id>
          <phase>package</phase>
          <goals>
            <goal>copy</goal>
          </goals>
          <configuration>
            <artifactItems>
              <artifactItem>
                <groupId>com.ontotext.graphdb</groupId>
                <artifactId>graphdb-se</artifactId>
                <version>${graphdb.version}</version>
                <type>zip</type>
                <classifier>dist</classifier>
                <outputDirectory>target/se-dist</outputDirectory>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

3.6.3 GraphDB JAR file for embedding the database or plugin development

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:

```
<dependency>
  <groupId>com.ontotext.graphdb</groupId>
  <artifactId>graphdb-enterprise-runtime</artifactId>
  <version>${graphdb.version}</version>
</dependency>
```

```
<dependency>
  <groupId>com.ontotext.graphdb</groupId>
  <artifactId>graphdb-standard-runtime</artifactId>
  <version>${graphdb.version}</version>
</dependency>
```

```
<dependency>
  <groupId>com.ontotext.graphdb</groupId>
  <artifactId>graphdb-enterprise-client</artifactId>
```

```
<version>${graphdb.version}</version>
</dependency>
```

3.6.4 I want to proxy the repository in my nexus

The following options have been verified to work as expected:

The screenshot shows the 'Onto Proxy' configuration window with the 'Configuration' tab selected. The window has several sections with various settings:

- Repository ID:** onto-proxy
- Repository Name:** Onto Proxy
- Repository Type:** proxy
- Provider:** Maven2
- Format:** maven2
- Repository Policy:** Release
- Default Local Storage Location:** file:/sonatype-work/storage/onto-proxy
- Override Local Storage Location:** (empty)
- Remote Repository Access:**
 - Remote Storage Location:** http://maven.ontotext.com/content/groups/ail-onto/
 - Download Remote Indexes:** False
 - Auto Blocking Enabled:** False
 - File Content Validation:** True
 - Checksum Policy:** Warn
- Access Settings:**
 - Allow File Browsing:** False
 - Include in Search:** False
 - Publish URL:** True
- Expiration Settings:**
 - Not Found Cache TTL:** 1440 minutes

At the bottom right, there are 'Save' and 'Reset' buttons.

Note that this won't give you the ability to browse and index/search our repository. This is a shortcoming of the fact that we achieve the authentication through a repository target at the moment.

!! [CHECK IF THE ABOVE IS STILL RELEVANT]

ADMINISTRATION

4.1 Administration tasks

The goal of this guide is to help you perform all common administrative tasks needed to keep a GraphDB database operational. These tasks include configuring the database, managing memory and storage, managing users, managing repositories, performing basic troubleshooting, creating backups, performance monitoring activities, and more.

The common administration tasks are:

- *Installation*
- *Configuring GraphDB*
- *Setting up Licenses*
- *Creating Locations*
- *Creating a Repository*
- *Configuring a Repository*
- *../access-rights-and-security*
- *Application Settings*
- *Backing up and Restoring a Repository*
- *Query Monitoring and Termination*
- *Troubleshooting*
 - *Database health checks*
 - *System metrics monitoring*
 - *Diagnosing and reporting critical errors*
 - *Storage tool*

4.2 Administration Tools

GraphDB can be administered through the *Workbench*, the JMX interface, or programmatically.

What's in this document?

- *Workbench*
- *JMX interface*
 - *Configuring the JMX endpoint*

4.2.1 Workbench

The Workbench is the web-based *administration interface* to GraphDB. It lets you administer GraphDB, as well as load, explore, manage, query and export data. To use it, start GraphDB in a workbench mode and open <http://localhost:7200/> in your browser.

4.2.2 JMX interface

After initialization, GraphDB registers a number of JMX MBeans for each repository, each providing a different set of information and functions for specific features.

4.2.2.1 Configuring the JMX endpoint

Configure the JMX endpoint using special system properties when starting the Java virtual machine (JVM), in which GraphDB is running. For example, the following command line parameters set the JMX server endpoint to listen on port 2815 without authentication and a secure socket layer:

- **Linux/Mac** - add the following configuration in `<graphdb_distribution>/bin/graphdb.in.sh`.

```
JAVA_OPTS_ARRAY+=("-Djava.rmi.server.hostname=`hostname`")
JAVA_OPTS_ARRAY+=("-Dcom.sun.management.jmxremote")
JAVA_OPTS_ARRAY+=("-Dcom.sun.management.jmxremote.port=2815")
JAVA_OPTS_ARRAY+=("-Dcom.sun.management.jmxremote.ssl=false")
JAVA_OPTS_ARRAY+=("-Dcom.sun.management.jmxremote.authenticate=false")
```

- **Windows** - add the following configuration in `<graphdb_distribution>/bin/graphdb.in.cmd`.

```
set JAVA_OPTS=%JAVA_OPTS% -Djava.rmi.server.hostname=`hostname`
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote.port=2815
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote.ssl=false
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote.authenticate=false
```

Once GraphDB is loaded, use any compliant JMX client, e.g., JConsole that is part of the Java development kit, to access the JMX interface on the configured port.

4.3 Setting up Licenses

What's in this document?

- *Setting up licenses through the Workbench*
- *Setting up licenses through a file*
 - *Custom file path property*
- *Order of preference*
- *Deprecated methods*

GraphDB SE is available under an RDBMS-like commercial license on a per-server-CPU basis. It is neither free nor open-source. To purchase a license or obtain a copy for evaluation, please contact GraphDB-info@ontotext.com.

When installing GraphDB SE, the license file can be set through the GraphDB Workbench or programmatically:

4.3.1 Setting up licenses through the Workbench

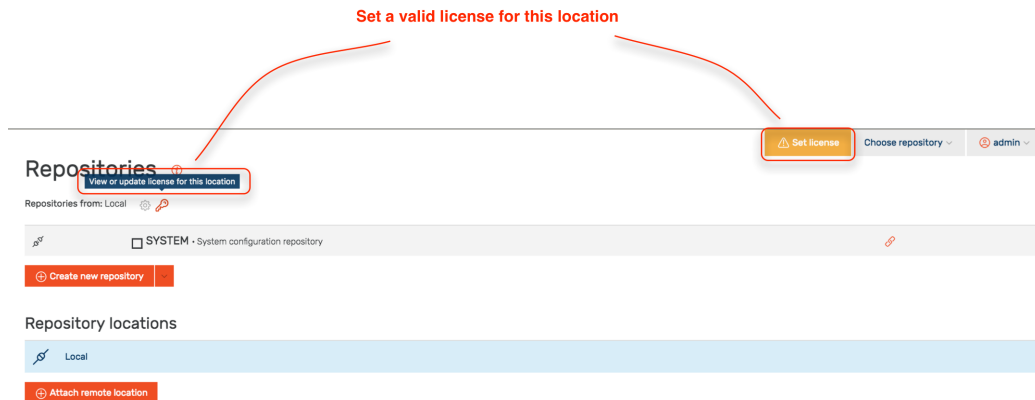
No license was set
Please supply a valid license or contact Ontotext AD sales department to acquire one.

[Set new license](#)

The software is provided 'AS IS', without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

To do that, follow the steps:

1. Add, view or update your license by clicking on the key icon next to the active location path.



2. Select the license file and register it.

Register GraphDB License ?

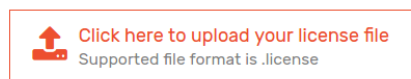
[Click here to upload your license file](#)
Supported file format is .license

or copy your license code below

[Register](#) [Cancel](#)

You can also copy and paste it in the text area.

Register GraphDB License ?



or copy your license code below

```
cC/cSSw8Xae8wQt++3Ah756UAJX50YxHVyDLwKa+Jdjnl8ZMjqin+1CUmiuPSTfcq7Be4plrqe1H2IT
Hv9v3TVaU6SJEK+adsad73854adsas98456jklsdfEE+0CtJnV0e4y2pl2dukvUatNN91ws4KaQlra+E1p
3VKkm2ITrd6PtxwIS58I0HLIPPbbJ2SvSCLZUyg2Flx2/E2YV7vAUxMDrAtwKWINKzXrMJ1lun9MRDwL
uswoC0QxbUMtyUgZXYveo6cMloot30FU9in4d0sTmsidufosnGI/+EfAet0+HiRouFN/h+PeXRyvlrKS
2tuouQ6Lsm209EILpeTijyM1zQ/xQ/B79rJGixMhHuCF970LR0VjWN2NnvajKSotUnmzn0tCwazz1f2k
LBfY2t/m305qQ4i6TpCKolGkQfGg5o6ha+f0p3G74UNqk03sgAcp7gKKXt9ntYvr6W4Ayo7ii6H87je
SMgNyCq9645ERZfwqLS9iodwbRsgf0U6CcJ1ia2u0e4kki+b89YbxIT4F0libQXuMvIwdCibazuBM3d+
eqA01uawIYk0BD2SjPLmXyjM1hMaXJlr5XGfD/Hyhys=
```

Register Cancel

3. Validate your license.

Are your license details correct? ×

Licensed to

Ontotext Internal - allowed only for internal development and testing

Valid until

2017-12-31

Number of cores

Unlimited

Non-commercial research only

Cancel

Yes

4. After completing these steps, you will see the details of your license.

4.3.2 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`.

4.3.2.1 Custom file path property

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

```
graphdb.license.file = /opt/graphdb/my-graphdb-dev.license
```

The license file must be readable by the user running GraphDB.

Note: If you set the license through a file in the `config` directory or a custom path, you will not be able to change the license through the GraphDB Workbench.

4.3.3 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 file `graphdb.license` in the configuration directory;
- A license set through the GraphDB Workbench.

4.3.4 Deprecated methods

GraphDB SE 9.0 supports some deprecated methods for setting the license. If you use one of them, please consider switching to one of the official methods, as support for the deprecated ones may be dropped in a future version. The deprecated methods are:

- The `OWLIM_LICENSE` environment variable;
- The `owlim-license` system property;
- The `owlim:owlim-license` repository property.

Note: The licenses for each commercial GraphDB edition are different, and will not work if used with the wrong software.

Tip: By default, GraphDB can be run on a machine with more CPUs than licensed.

4.4 Creating Locations

What's in this document?

- *Active location*
- *Inactive location*
- *Connect to a remote location*
- *Configure a data location*

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. Only a single location can be active at a time.

To manage your data locations:

1. Start a browser and go to the Workbench web application using a URL of this form: <http://localhost:7200>.
- substituting `localhost` and the `7200` port number as appropriate.
2. Go to *Setup -> Repositories*.

4.4.1 Active location

When started, GraphDB creates `GraphDB-HOME/data` directory as an active location. To change the directory, see *Configuring GraphDB Data Directory*.

Repositories

Repositories from: Local  

  rank     

 Create new repository 

Repository locations

 Local

 Attach remote location

Change active location settings

By default, the active location does not send anonymous usage statistics to Ontotext. To change this, click on the icon *Change active location settings* and enable it.



Settings



☐ Send anonymous usage statistics to Ontotext

[less info](#) ^

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 plugin is used, e.g. geo-spatial
 - Ruleset (custom rulesets are reported only as "custom")
 - OS 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.

Cancel

Save settings

View or update its license

Click the *Key* icon



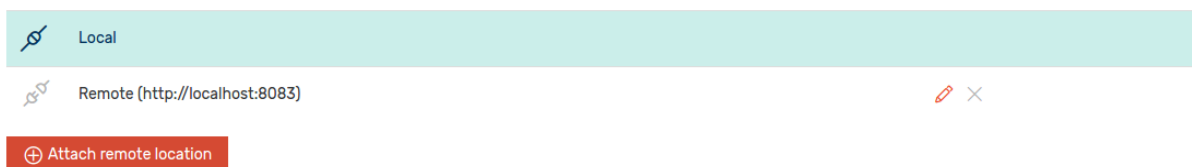
to check the details of your current license.

Licensed to	Valid until	Number of cores
Ontotext Internal - allowed only for internal development and testing	2017-12-31	Unlimited
Non-commercial research only		

4.4.2 Inactive location

All inactive locations are listed below the active repository window. Here, you can change the locations settings, as well as disconnect the location from the running GraphDB.

Repository locations



4.4.3 Connect to a remote location

To connect to a remote location:

1. Click the **Connect to location** button and add the HTTP RDF4J Location of your repository, for example `http://localhost:8083`.

Connect to a remote GraphDB instance ×

Location URL

http://localhost:8083/

Enter a URL to a remote GraphDB instance

Hide advanced options ^

Optional

Username

admin

Password

....

Superadmin settings

Jolokia secret

secret

Cancel

⊕ Add

- Optionally, you can set a user and a password for this location.

You can attach multiple locations but only one can be active at a given time. The active location is always shown in the navigation bar next to a plug icon.

Note: Using basic HTTP authentication may be required for accessing the HTTP-JMX bridge in the Monitor views.

Note: If you use the Workbench as a SPARQL endpoint, all your queries are sent to a repository in the *currently active* location. This works well if you do not change the active location. To have endpoints that are always accessible outside the Workbench, we recommend using standalone Workbench and Engine installations, connecting the Workbench to the Engine over a remote location and using the Engine endpoints (i.e., not the ones provided by the Workbench) in any software that executes SPARQL queries.

Note: You can connect to a remote location over HTTPS as well. In order to do so, you need to: 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 your JVM's SSL TrustStore.

4.4.4 Configure a data location

Set the property `graphdb.home.data` in `<graphdb_dist>/conf/graphdb.properties`. If no property is set, the default repositories location will be: `<graphdb_dist>/data`.

4.5 Creating a Repository

What's in this document?

- *Create a repository*
 - *Using the Workbench*
 - *Using the RDF4J console*
- *Manage repositories*
 - *Select a repository*
 - *Make it a default repository*
 - *Edit a repository*

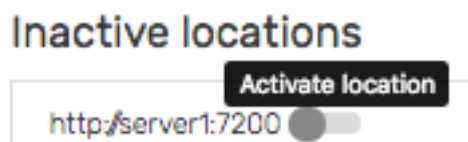
4.5.1 Create a repository

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

4.5.1.1 Using the Workbench

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

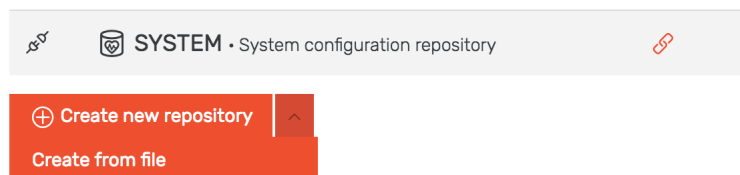
1. Click the *Connect* button next to the location you want to activate.



2. Click the *Create new repository* button or create it from a file by *using the configuration template* that can be found at `configs/templates/`.

Repositories ⓘ

Repositories from: Local ⚙️ 🔑



3. Enter the *Repository ID* (e.g., `repository1`) 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*.

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

4.5.1.2 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:

```
console.cmd (Windows)
./console (Unix/Linux)
```

2. Connect to the GraphDB server instance using the command:

```
connect http://localhost:7200.
```

3. Create a repository using the command:

```
create se.
```

4. Fill in *the values of the parameters* in the console.

5. Exit the RDF4J console:

```
quit.
```

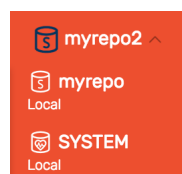
4.5.2 Manage repositories

4.5.2.1 Select a repository

- Connect the newly created repository to the active location.

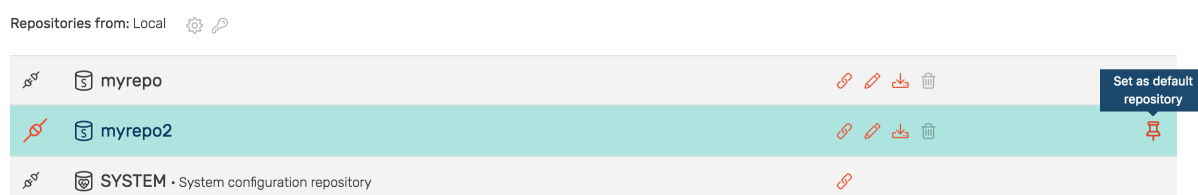


- Alternatively, use the dropdown menu in the top right corner. This allows you to easily change the repository while running queries or importing and exporting data in other views.



4.5.2.2 Make it a default repository

Use the pin to select it as a default repository.



4.5.2.3 Edit a repository

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



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

4.6 Configuring a Repository

Before you start adding or changing the parameter values, it is good to plan your repository configuration, to know 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 set up, etc.

What's in this document?

- *Plan a repository configuration*
- *Configure a repository through the GraphDB Workbench*
- *Edit a repository*
- *Configure a repository programmatically*
- *Configuration parameters*
- *Configure GraphDB memory*
 - *Configure Java heap memory*
 - *Single global page cache*
 - *Configure Entity pool memory*
 - *Sample memory configuration*
 - *Upper bounds for the memory consumed by the GraphDB process*
- *Reconfigure a repository*
 - *Using the Workbench*
 - *Global overrides*
- *Rename a repository*
 - *Using the workbench*

4.6.1 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*

4.6.2 Configure a repository through the GraphDB Workbench

To configure a new repository, complete the repository properties form.

Create Repository

Repository properties

Repository ID*	This field is required
Repository title	
Type	GRAPHDB-SE
Storage folder	storage
Ruleset	RDFS-Plus (Optimized) Upload custom ruleset
☒ Disable owl:sameAs	
Base URL	http://example.org/owlim#
Entity index size	10000000
☒ Use predicate indices ☒ Cache literal language tags	
☐ Use context index ☒ Enable literal index	
☐ Check for inconsistencies ☐ Throw exception on query time-out	
☐ Read-only	
Entity ID bit-size	32
Query time-out (seconds)	0
Limit query results	0
Create Cancel	

4.6.3 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.

4.6.4 Configure a repository programmatically

Tip: GraphDB uses an [RDF4J configuration template](#) for configuring its repositories. RDF4J keeps the repository configurations with their parameters, modelled in RDF. Therefore, in order to create a new repository, the 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 `/templates` folder of the GraphDB distribution. The parameters are described in the [Configuration parameters](#) section.

```
# RDF4J configuration template for a GraphDB Standard Edition 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 owl: <http://www.ontotext.com/tree/owlim#>.

[] a rep:Repository ;
  rep:repositoryID "graphdb-test" ;
  rdfs:label "GraphDB SE repository" ;
  rep:repositoryImpl [
    rep:repositoryType "owlim:MonitorRepository" ;
    sr:sailImpl [
      sail:sailType "owlim:Sail" ;

      owl:base-URL "http://example.org/graphdb#" ;
      owl:defaultNS "" ;
      owl:entity-index-size "10000000" ;
      owl:entity-id-size "32" ;
      owl:imports "" ;
      owl:repository-type "file-repository" ;
      owl:ruleset "rdfsplus-optimized" ;
      owl:storage-folder "storage" ;

      owl:enable-context-index "false" ;

      owl:enablePredicateList "true" ;

      owl:in-memory-literal-properties "true" ;
      owl:enable-literal-index "true" ;

      owl:check-for-inconsistencies "false" ;
      owl:disable-sameAs "false" ;
      owl:query-timeout "0" ;
      owl:query-limit-results "0" ;
      owl:throw-QueryEvaluationException-on-timeout "false" ;
      owl:read-only "false" ;
      owl:nonInterpretablePredicates "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" ;
    ]
  ].
```

2. Use this command to create a repo from a `config.ttl`:

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

4.6.5 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).

base-URL

Description: 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.

Default value: none

Can be changed.

check-for-inconsistencies (*see more*)

Description: 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 detected while committing a transaction, the whole transaction will be rolled back.

Default value: false

Can be changed.

defaultNS

Description: 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.

Default value: <empty>

Example: `owlim:defaultNS "http://www.w3.org/2002/07/owl#;http://example.org/owlim#".`

Cannot be changed.

Warning: This parameter cannot be set via a command line argument.

disable-sameAs (*see more*)

Description: Enables or disables the `owl:sameAs` optimization.

Default value: false

Warning: This parameter needs special attention.

enable-context-index (*see more*)

Default value: false

Possible value: true, where GraphDB will build and use the context index.

Can be changed.

enable-literal-index (*see more*)

Description: 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.

Default value: true

Can be changed.

enablePredicateList (*see more*)

Description: 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.

Default value: false

Can be changed.

entity-id-size

Description: Defines the bit size of internal IDs used to index entities (URIs, blank nodes and literals). In most cases, 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. 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.

Default value: 32

Possible values: 32 and 40

Cannot be changed.

entity-index-size *(see more)*

Description: Defines the **initial size** of the entity hash table index entries. The bigger the size, the fewer the collisions in the hash table, and the faster the entity retrieval. The entity hash table will adapt to the number of stored entities once the number of collisions passes a critical threshold.

Default value: 10,000,000

Warning: This parameter cannot be changed by the user, once initially set.

imports

Description: A list of schema files that will be imported at startup. All statements found in these files will be loaded in the repository and will be treated as read-only. The serialization format is determined by the file extension:

- .brf => BinaryRDF
- .n3 => N3
- .nq => N-Quads
- .nt => N-Triples
- .owl => RDF/XML
- .rdf => RDF/XML
- .rdfs => RDF/XML
- .trig => TriG
- .trix => TriX
- .ttl => Turtle
- .xml => TriX

Default value: none

Example: `owl:imports "/ont/owl.rdfs;./ont/ex.rdfs"`.

Cannot be changed.

Tip: Schema files can be either a local path name, e.g., `./ontology/myfile.rdf` or a URL, e.g., `http://www.w3.org/2002/07/owl.rdf`. If this parameter is used, the default namespace for each imported schema file **must** be provided using the *defaultNS* parameter.

in-memory-literal-properties *(see more)*

Description: Enables or disables caching of the literal languages and data types. If the caching is on and the entity pool is restored from persistence, but there is no such cache available on disk, it is created after the entity pool initialization.

Default value: false

Can be changed.

nonInterpretablePredicates

Description: Colon-separated list of predicates (full URLs) that GraphDB will not try to process with the registered GraphDB plugins. (Predicates processed by registered plugins are often called “Magic” predicates). This optimization will speed up the data loading by providing a hint that these predicates are not magic.

Default 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`

query-limit-results

Description: Sets the maximum number of results returned from a query after which the evaluation of a query will be terminated; values less than or equal to zero mean no limit.

Default value: 0; (no limit);

Can be changed.

query-timeout (*see more*)

Description: Sets the number of seconds after which the evaluation of a query will be terminated; values less than or equal to zero mean no limit.

Default value: 0; (no limit);

Can be changed.

read-only

Description: In this mode, no modifications are allowed to the data or namespaces.

Default value: false

Possible value: true, puts the repository in read-only mode.

Can be changed.

repository-type

Default value: file-repository

Possible values: file-repository, weighted-file-repository.

Cannot be changed.

ruleset (*see more*)

Description: Sets of axiomatic triples, consistency checks and entailment rules, which determine the applied semantics.

Default value: rdfs-plus-optimized

Possible values: empty, rdfs, owl-horst, owl-max and owl2-rl and their optimized counterparts rdfs-optimized, owl-horst-optimized, owl-max-optimized and owl2-rl-optimized. A custom ruleset is chosen by setting the path to its rule file .pie.

Tip: *Hints on optimizing GraphDB's rulesets.*

Warning: This parameter needs special attention.

storage-folder (*see more*)

Description: specifies the folder where the index files will be stored.

Default value: none

Can be changed.

throw-QueryEvaluationException-on-timeout

Default value: false

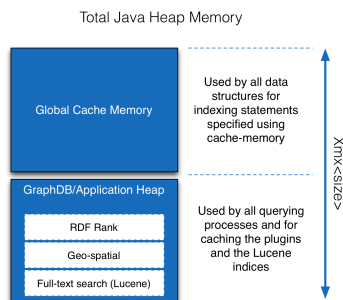
Possible value: true; if set, a `QueryEvaluationException` is thrown when the duration of a query execution exceeds the timeout parameter.

Can be changed.

4.6.6 Configure GraphDB memory

4.6.6.1 Configure Java heap memory

The following diagram offers a view of the memory use by the GraphDB structures and processes:



To specify the maximum amount of heap space used by a JVM, use the `-Xmx` virtual machine parameter.

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`.

4.6.6.2 Single global page cache

GraphD'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 don't specify `page.cache.size` but only enable the global cache, it will take 50% of the `-Xmx` parameter.

Note: You don't have to change/edit your repository configurations. The new cache will be used when you upgrade to the new version.

4.6.6.3 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).

4.6.6.4 Sample memory configuration

This is a sample configuration demonstrating how to correctly size a GraphDB server with a single repository. The loaded dataset is estimated to 500 million RDF statements and 150 million unique entities. As a rule of thumb, the average number of unique entities compared to the total number of statements in a standard dataset is 1:3.

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

4.6.6.5 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.

4.6.7 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.

4.6.7.1 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.

4.6.7.2 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.

4.6.8 Rename a repository

4.6.8.1 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.

4.7 Secure GraphDB

What's in this document?

- *Enable security*
- *Login and default credentials*
- *Free access*
- *Users and Roles*
 - *Create new user*
 - *Set password*

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.

4.7.1 Enable security

Users and Access ⓘ

 Create new user

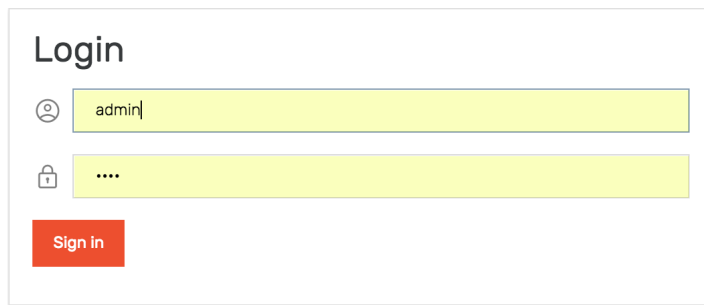
Security is **OFF** 

Username	Role	Repository rights	Date created	Actions
admin	Administrator	Unrestricted	2018-06-15 09:29:18	

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.

4.7.2 Login and default credentials



The login form is titled "Login". It contains two input fields: the first is for the username, with "admin" entered; the second is for the password, shown as four dots. Below the password field is a red button labeled "Sign in".

The default admin credentials are:

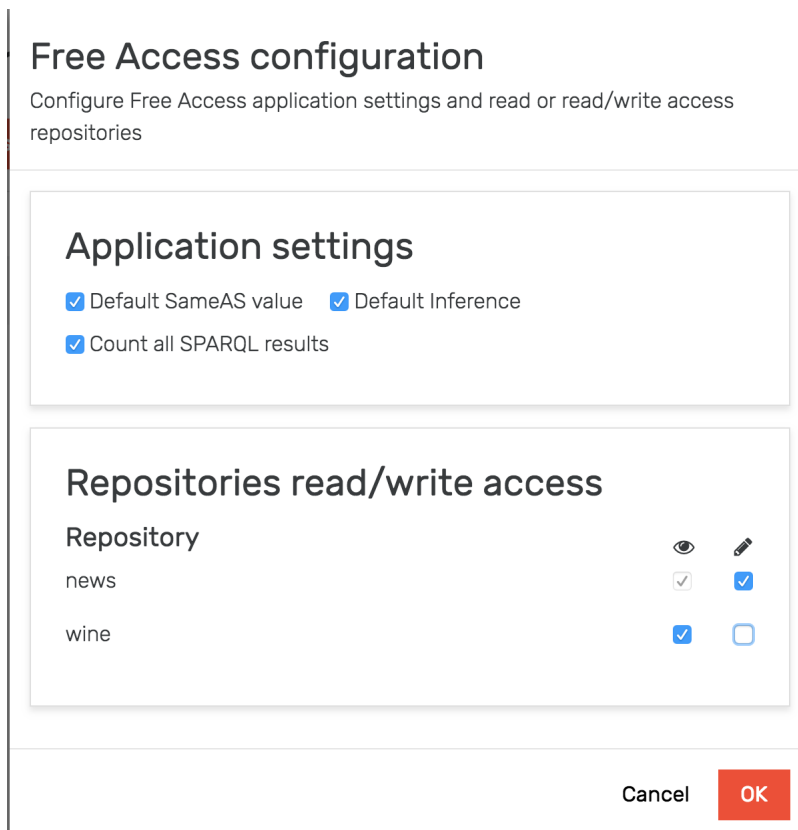
username: **admin**

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.

4.7.3 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:



The "Free Access configuration" dialog box has a title bar and a subtitle: "Configure Free Access application settings and read or read/write access repositories". It contains two main sections:

- Application settings:**
 - ☒ Default SameAS value
 - ☒ Default Inference
 - ☒ Count all SPARQL results
- Repositories read/write access:**

Repository	Read Access (Eye Icon)	Write Access (Pencil Icon)
news	☒	☒
wine	☒	☐

At the bottom right, there are "Cancel" and "OK" buttons.

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 “news”, and read the “wine” repository. They will also be able to create or delete connectors and toggle plugins for the “news” repository.

Application settings allow you to configure the default behavior for the GraphDB Workbench.

4.7.4 Users and Roles

4.7.4.1 Create new user

This is the user creation screen.

news

admin

Create new user

Login

User name

Password

Confirm password

User role

☒ User
 ☐ Repository manager
 ☐ Administrator

Repository rights

Repository		
Any data repository ⓘ	☒	☐
news	☒	☐
wine	☒	☒

SPARQL editor settings

Expand results over owl:SameAs is ☒ ON ☐ by default

Inference is ☒ ON ☐ by default

☒ Count total results

☐ Ignore shared saved queries

Create

Cancel

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.

Note: If you need to speed up a slow repository with enabled security when each request includes HTTP basic authentication, we recommend some [options here](#).

4.7.4.2 Set password

Edit user: admin

news admin

Login

User role

☐ User
 ☐ Repository manager
 ☒ Administrator

Repository rights

Repository		
Any data repository	☒	☒
news	☒	☒
wine	☒	☒

SPARQL editor settings

Expand results over owl:SameAs is ☒ ON ☐ by default

Inference is ☒ ON ☐ by default

☒ Count total results

☐ Ignore shared saved queries

Save Cancel

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

4.8 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
```

In a cluster, headers will be logged in the masters' logs and the logs of the worker executing the request.

4.9 Application Settings

Application settings help you to configure the default behaviour 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:

Application settings

- ☒ SPARQL Editor: Expand results over owl:SameAs is ON by default
- ☒ SPARQL Editor: Inference is ON by default
- ☒ SPARQL Editor: Count all results

Save

Cancel

- 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.
- Default Inference value - Same as above, but for the *Include inferred data in results* option in the SPARQL editor.
- 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.

Application settings are user-based. When security is ON, each user can access his/her 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 set the application settings.

4.10 Backing up and Restoring a Repository

What's in this document?

- *Back up a repository*
 - *Export repository to an RDF file*
 - *Back up a repository using JMX interface*
 - *Back up GraphDB by copying the binary image*
- *Restore a repository*

4.10.1 Back up a repository

Repository backups allow users 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.

4.10.1.1 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:

```
RepositoryConnection connection = repository.getConnection();
FileOutputStream outputStream = new FileOutputStream(new File("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.

4.10.1.2 Back up a repository using JMX interface

GraphDB offers backing up a repository through JMX.

The `OwlimRepositoryManager` MBean is extended through the `createZipBackup(String folder)` method. Invocation with a folder argument creates a zip file named `rep_<repository_id>-<timestamp>_backup.zip` with the content of the repository data dir (storage folder is subfolder there, so the `config.ttl` should be archived too). If you add `backupName` for directory, backup will be created in the `<repository_id>/backup/<backupName>` folder.

Invoking method with `null` parameter for Backup name will result in creating backup in the `<repository_id>/backup/default` folder.

Note: Any attempt to create backup with an invalid Backup name 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:

The screenshot shows the JMX console interface. On the left, the 'Overview' tab is active, displaying a tree of MBeans. The path 'com.ontotext.OwlimRepositoryManager>Operations>createZipBackup' is highlighted. On the right, the 'Operation invocation' section shows the signature 'void createZipBackup (p1 String)'. Below it, the 'MBeanOperationInfo' table lists details for the 'createZipBackup' operation.

Name	Value
Operation:	
Name	createZipBackup
Description	Operation exposed for management
Impact	UNKNOWN
ReturnType	void
Parameter-0:	
Name	p1
Description	
Type	java.lang.String

Below the MBeanOperationInfo table is a 'Descriptor' section with a table structure:

Name	Value

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.

4.10.1.3 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 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](#).

4.10.2 Restore a repository

The restore options depends 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 > RDF*, 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. Stop the GraphDB server.
2. Delete the entire `your-repo` folder, and copy/paste the folder of the `{graphdb.home.data}/repositories/your-repo` from the backup copy.
3. Start the GraphDB server.
4. Run a quick test read query to make sure that the repository is initialized correctly.

4.11 Query Monitoring and Termination

What's in this document?

- [Query monitoring and termination using the workbench](#)
- [Query monitoring and termination using the JMX interface](#)
 - [Query monitoring](#)
 - [Terminating a query](#)
- [Terminating a transaction](#)
- [Automatically prevent long running queries](#)

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.

4.11.1 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.

Query and Update monitoring ⓘ

1 query/update

news ▾

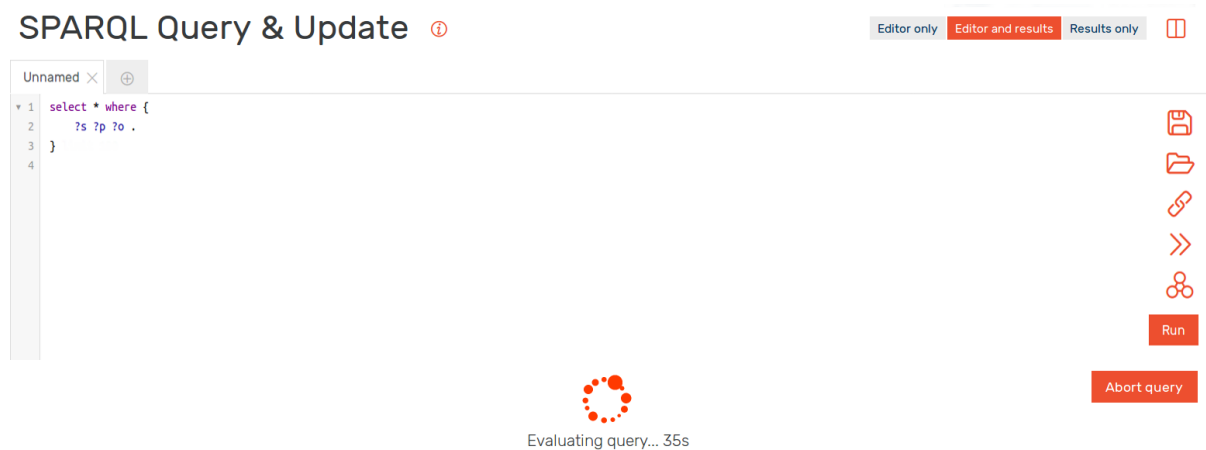
Pause

id	node	type	query	lifetime	state	
13	news	UPDATE	Download <pre> insert { inst:allNews :createIndex "-termweight idf -minfrequency 3" ; :documentID ?documentID . ?documentID :documentText ?documentText . } where { SELECT ?documentID ?documentText { ?documentID pubo:content ?documentText . FILTER(isLiteral(?documentText)) } } </pre>	1m 18s	IN_COMMIT_PLUGIN 435858 operations	Abort update

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

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

You can also interrupt a query directly from the SPARQL Editor:



4.11.2 Query monitoring and termination using the JMX interface

4.11.2.1 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.

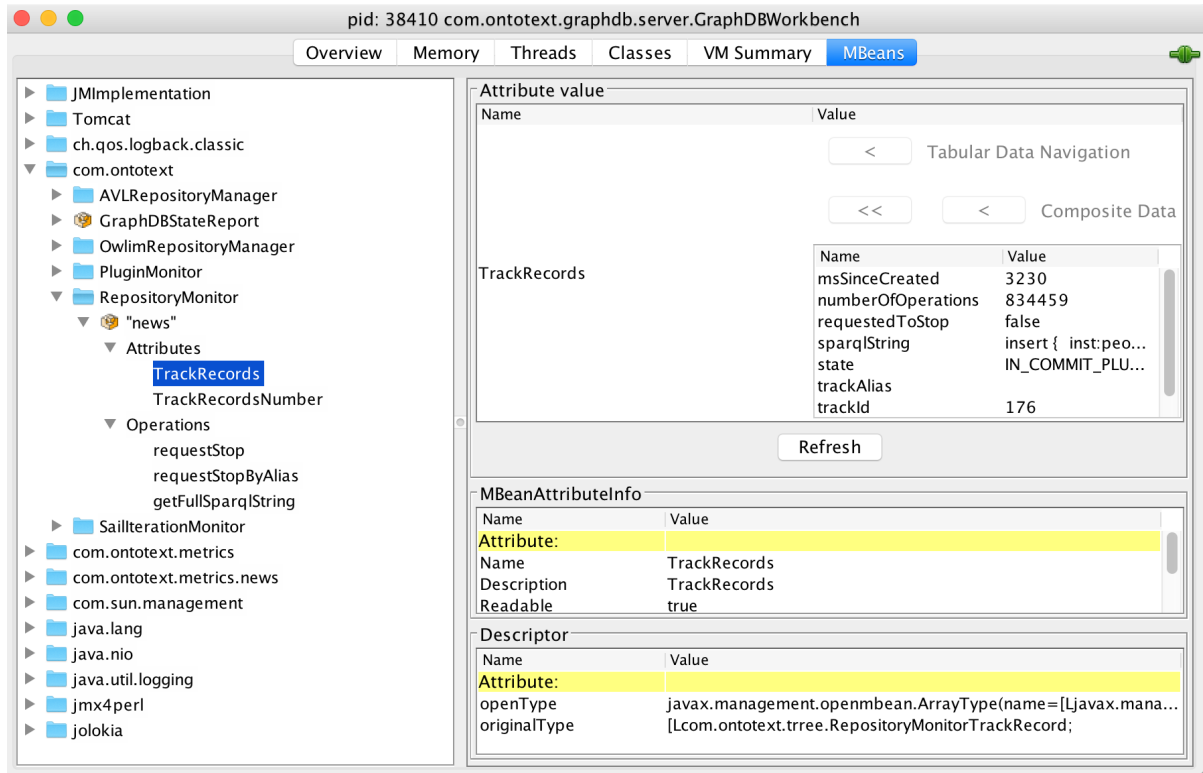
Package	com.ontotext
MBean name	RepositoryMonitor

The RepositoryMonitor MBean has two attributes - TrackRecords and TrackRecordsNumber.

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

Attribute	Description
msSinceCreated	the time (in ms) since the iterator was created
numberOfOperations	the total number of operations for this iterator
requestedToStop	indicates if the query has been requested to terminate early (see below)
sparqlString	the full text of the query
state	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
trackAlias	a unique alias for this iterator given by the client
trackId	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
type	the type of this iterator - UPDATE or READ

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.

4.11.2.2 Terminating a query

The following operations are available with RepositoryMonitor MBean:

Operation	Description
requestStop	Requests that a query terminates early; parameter: trackId of the query to stop
requestStopByAlias	Requests that a query terminates early; parameter: trackAlias of the query to stop
getFullSparqlString	Returns the full text of the query; parameter: trackId of the query

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.

4.11.3 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:

Package	com.ontotext
MBean name	OwlimRepositoryManager

This MBean has no attributes:

Operation	Description
abortTransaction-Commit	Requests that the currently executing (lengthy) commit operation be terminated and rolled back.

4.11.4 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.

4.12 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 indices*
 - *Cache/index monitoring and optimizations*
 - *Query optimizations*
- *Explain Plan*
- *Inference optimizations*
 - *Delete optimizations*
 - *Rules optimizations*
 - *Optimization of owl:sameAs*
 - *RDFS and OWL support optimizations*

4.12.1 Data loading & query optimizations

What's in this document?

- *Dataset loading*
 - *Normal operation*
- *GraphDB's optional indices*
 - *Predicate lists*
 - *Context index*
- *Cache/index monitoring and optimizations*
- *Query optimizations*
 - *Caching literal language tags*
 - *Not enumerating sameAs*

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.

4.12.1.1 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 indices:

```
enablePredicateList = false (unless the dataset has a large number of predicates)
enable-context-index = false
in-memory-literal-properties = false
```

Normal operation

The size of the data structures used to index entities is directly related to the number of unique entities in the loaded dataset. These data structures are always kept in memory. In order to get an upper bound on the number of unique entities loaded and to find the actual amount of RAM used to index them, it is useful to know the contents of the storage folder.

The total amount of memory needed to index entities is equal to the sum of the sizes of the files `entities.index` and `entities.hash`. This value can be used to determine how much memory is used and therefore how to divide the remaining memory between the cache memory, etc.

An upper bound on the number of unique entities is given by the size of `entities.hash` divided by 12 (memory is allocated in pages and therefore the last page will likely not be full).

The file `entities.index` 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:

entity-index-size (*see more*) Set to a large enough value.

enablePredicateList (*see more*) Can speed up queries (and loading).

enable-context-index (*see more*) Provides better performance when executing queries that use contexts.

index-in-memory-literal-properties (*see more*) Defines whether to keep the properties of each literal in-memory.

Furthermore, the inference semantics can be adjusted by choosing a different ruleset. However, this will require a reload of the whole repository, otherwise some inferences may remain in the wrong location.

Note: The optional indices 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.

4.12.1.2 GraphDB's optional indices

Predicate lists

Predicate lists are two indices (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 indices for both loading and query answering. Predicate list indices 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.

4.12.1.3 Cache/index monitoring and optimizations

Statistics are kept for the main index data structures, and include information such as cache hits/misses, file reads/writes, etc. This information can be used to fine-tune GraphDB memory configuration, and can be useful for 'debugging' certain situations, such as understanding why load performance changes over time or with particular data sets.

The screenshot shows the JMX console with the following structure:

- com.ontotext.metrics
 - cache
 - Count: 79712
 - cache.flush
 - Count: 0
 - cache.load
 - Count: 0
 - cache.miss
 - Count: 0
 - cache.write
 - Count: 0

The 'Attribute values' table on the right shows the following data:

Name	Value
Count	79712

For each index, there will be a `CollectionStatistics` MBean published, which shows the cache and file I/O values updated in real time:

Package	com.ontotext
MBean name	CollectionStatistics

The following information is displayed for each MBean/index:

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

The following operations are available:

Operation	Description
resetCounters	Resets all the counters for this index.

Ideally, the system should be configured to keep the number of cache misses to a minimum. If the ratio of hits to misses is low, consider increasing the memory available to the index (if other factors permit this).

Page swaps tend to occur much more often during large scale data loading. Page discards occur more frequently during query evaluation.

4.12.1.4 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 ( 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:

```
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/Mx4ruQS1AL_QQdeZXf-MIWWdng
umbel-sc:CommercialAirport
opencyc:Mx4ruQS1AL_QQdeZXf-MIWWdng
dbpedia:Jetport
dbpedia:Airstrips
dbpedia:Airport
fb:guid.9202a8c04000641f800000000004ae12
opencyc-en:CommercialAirport
```

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

```
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:

```
dbpedia:Air_strip
opencyc-en:CommercialAirport
```

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.

4.12.2 Explain Plan

What's in this document?

- *What is GraphDB's Explain Plan*
- *Activating the explain plan*
- *Simple explain plan*
- *Multiple triple patterns*
- *Wine queries*
 - *First query with aggregation*

4.12.2.1 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.

4.12.2.2 Activating the explain plan

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

```
PREFIX onto: <http://www.ontotext.com/>
select * from onto:explain
...
```

4.12.2.3 Simple explain plan

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

```
PREFIX onto: <http://www.ontotext.com/>
select * from onto:explain {
  ?s ?p ?o .
}
```

Depending on the number of triples that you have in the database, the results will vary, but you will get something like the following:

```
SELECT ?s ?p ?o
{
  { # ----- Begin optimization group 1 -----
    ?s ?p ?o . # Collection size: 108.0
               # Predicate collection size: 108.0
               # Unique subjects: 90.0
               # Unique objects: 55.0
               # Current complexity: 108.0
  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 108.0
}
```

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 optimisation 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: 108.0 - the approximate number of iterations that will be executed for this group.

4.12.2.4 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:

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

select *
from onto:explain
{
    ?o rdf:type ?o1 .
    ?o rdfs:subPropertyOf ?o2
}
```

and the output is:

```
SELECT ?o ?o1 ?o2
{
  { # ----- Begin optimization group 1 -----

    ?o rdfs:subPropertyOf ?o2 . # Collection size: 20.0
                                # Predicate collection size: 20.0
                                # Unique subjects: 19.0
                                # Unique objects: 18.0
                                # Current complexity: 20.0

    ?o rdf:type ?o1 . # Collection size: 43.0
                    # Predicate collection size: 43.0
                    # Unique subjects: 34.0
                    # Unique objects: 7.0
                    # Current complexity: 860.0

  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 25.294117647058822
}
```

Understanding the output:

- `?o rdfs:subPropertyOf ?o2` has a lower collection size (20 instead of 43), so it will be executed first.
- `?o rdf:type ?o1` has a bigger collection size (43 instead of 20), 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 20 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 $20 * 43 = 860$.
- Although the complexity for the whole group is 860, the estimated number of iterations for this group is 25.3.

4.12.2.5 Wine queries

All of the following examples refer to our simple wine dataset (*wine.ttl*). The file is quite small, but here is some basic explanation about the 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.

First query with aggregation

A typical aggregation query contains a group with some aggregation function. Here, we have added an explain graph:

```
# Retrieve the number of wines produced in each year along with the year
PREFIX onto: <http://www.ontotext.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.ontotext.com/example/wine#>
SELECT (COUNT(?wine) as ?wines) ?year
FROM onto:explain
WHERE {
    ?wine rdf:type :Wine .
    OPTIONAL {
        ?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):

```
SELECT (COUNT(?wine) AS ?wines) ?year
{
  { # ----- Begin optimization group 1 -----
    ?wine rdf:type :wine#Wine . # Collection size: 5.0
                                # Predicate collection size: 64.0
                                # Unique subjects: 50.0
                                # Unique objects: 12.0
                                # Current complexity: 5.0
  } # ----- End optimization group 1 -----
  # ESTIMATED NUMBER OF ITERATIONS: 5.0

  OPTIONAL
  {
    { # ----- Begin optimization group 2 -----
      ?wine :hasYear ?year . # Collection size: 5.0
                            # Predicate collection size: 5.0
                            # Unique subjects: 5.0
                            # Unique objects: 2.0
                            # Current complexity: 5.0
    } # ----- End optimization group 2 -----
    # ESTIMATED NUMBER OF ITERATIONS: 5.0
  }
}
GROUP BY ?year
```

```
ORDER BY DESC(?wines)
LIMIT 1000
```

4.12.3 Inference optimizations

4.12.3.1 Delete optimizations

What's in this document?

- *The algorithm*
- *Example*
- *Schema transactions*

GraphDB's *inference* policy is based on materialization, where implicit statements are inferred from explicit statements as soon as they are inserted into the repository, using the specified semantics ruleset. This approach has the advantage of achieving query answering very quickly, since no inference needs to be done at query time.

However, no justification information is stored for inferred statements, therefore deleting a statement normally requires a full re-computation of all inferred statements. This can take a very long time for large datasets.

GraphDB uses a special technique for handling the deletion of explicit statements and their inferences, called "*smooth delete*". It allows fast delete operations as well as ensures that schemas can be changed when necessary.

The algorithm

The algorithm for identifying and removing the inferred statements that can no longer be derived by the explicit statements that have been deleted, is as follows:

1. Use forward chaining to determine what statements can be inferred from the statements marked for deletion.
2. Use backward chaining to see if these statements are still supported by other means.
3. Delete explicit statements and the no longer supported inferred statements.

Note: We recommend that you mark the visited statements as read-only. Otherwise, as almost all delete operations follow inference paths that touch schema statements, which then lead to almost all other statements in the repository, the smooth delete can take a very long time. However, since a read-only statement cannot be deleted, there is no reason to find what statements are inferred from it (such inferred statements might still get deleted, but they will be found by following other inference paths).

Statements are marked as read-only if they occur in the Axioms section of the ruleset files (standard or custom) or are loaded at initialization time via the *imports configuration parameter*.

Note: When using 'smooth delete', we recommend that you load all ontology/schema/vocabulary statements using the imports configuration parameter.

Example

Consider the following statements:

```
Schema:
<foaf:name> <rdfs:domain> <owl:Thing> .
<MyClass> <rdfs:subClassOf> <owl:Thing> .
```

```
Data:
<wayne_rooney> <foaf:name> "Wayne Rooney" .
<Reviewer40476> <rdf:type> <MyClass> .
<Reviewer40478> <rdf:type> <MyClass> .
<Reviewer40480> <rdf:type> <MyClass> .
<Reviewer40481> <rdf:type> <MyClass> .
```

When using the owl-horst ruleset the removal of the statement:

```
<wayne_rooney> <foaf:name> "Wayne Rooney"
```

will cause the following sequence of events:

```
rdfs2:
x a y - (x=<wayne_rooney>, a=foaf:name, y="Wayne Rooney")
a rdfs:domain z (a=foaf:name, z=owl:Thing)
-----
x rdf:type z - The inferred statement [<wayne_rooney> rdf:type owl:Thing] is to be removed.
```

```
rdfs3:
x a u - (x=<wayne_rooney>, a=rdf:type, u=owl:Thing)
a rdfs:range z (a=rdf:type, z=rdfs:Class)
-----
u rdf:type z - The inferred statement [owl:Thing rdf:type rdfs:Class] is to be removed.
```

```
rdfs8_10:
x rdf:type rdfs:Class - (x=owl:Thing)
-----
x rdfs:subClassOf x - The inferred statement [owl:Thing rdfs:subClassOf owl:Thing] is to be removed.
```

```
proton_TransitiveOver:
y q z - (y=owl:Thing, q=rdfs:subClassOf, z=owl:Thing)
p protons:transitiveOver q - (p=rdf:type, q=rdfs:subClassOf)
x p y - (x=[<Reviewer40476>, <Reviewer40478>, <Reviewer40480>, <Reviewer40481>], p=rdf:type,
↪ y=owl:Thing)
-----
x p z - The inferred statements [<Reviewer40476> rdf:type owl:Thing], etc., are to be removed.
```

Statements such as [<Reviewer40476> rdf:type owl:Thing] exist because of the statements [<Reviewer40476> rdf:type <MyClass>] and [<MyClass> rdfs:subClassOf owl:Thing].

In large datasets, there are typically millions of statements [X rdf:type owl:Thing], and they are all visited by the algorithm.

The [X rdf:type owl:Thing] statements are not the only problematic statements considered for removal. Every class that has millions of instances leads to similar behavior.

One check to see if a statement is still supported requires about 30 query evaluations with owl-horst, hence the slow removal.

If [owl:Thing rdf:type owl:Class] is marked as an axiom (because it is derived by statements from the schema, which must be axioms), then the process stops when reaching this statement. So, the schema (the system statements) must necessarily be imported through the imports configuration parameter in order to mark the schema statements as axioms.

Schema transactions

As mentioned above, ontologies and schemas imported at initialization time using the *imports configuration parameter* configuration parameter are flagged as read-only. However, there are times when it is necessary to change a schema. This can be done inside a ‘system transaction’.

The user instructs GraphDB that the transaction is a system transaction by including a dummy statement with the special `schemaTransaction` predicate, i.e.:

```
_:b1 <http://www.ontotext.com/owlim/system#schemaTransaction> _:b2
```

This statement is not inserted into the database, but is rather serving as a flag telling GraphDB that the statements from this transaction are going to be inserted as read-only; all statements derived from them are also marked as read-only. When you delete statements in a system transaction, you can remove statements marked as read-only, as well as statements derived from them. Axiom statements and all statements derived from them stay untouched.

4.12.3.2 Rules optimizations

What’s in this document?

- *How to enable rule profiling*
 - *Log file*
 - *Excel format*
 - *Investigating performance*
- *Hints on optimizing GraphDB’s rulesets*
 - *Know what you want to infer*
 - *Minimize the number of rules*
 - *Write your rules carefully*
 - *Avoid duplicate statements*
 - *Know the implications of ontology mapping*
 - *Consider avoiding inverse statements*
 - *Consider avoiding long transitive chains*
 - *Consider specialized property constructs*
 - *Additional ruleset usage optimization*

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 don't have the required instrumentation (counters);
- The debug rules statistics are available only for importing data in **serial mode**. It doesn't work for Parallel Inferencing, which is default. Check **"Force serial pipeline"** in the Import settings dialog to enable it.

Warning: Rule profiling slows down the rule execution (the leading premise checking part) by 10-30%, so do not use it in production.

Log file

When rule profiling is enabled:

- Complete rule statistics are printed at every million statements, every 5 minutes, or on shutdown, depending on which occurs first.
- They are written to graphdb-folder/logs/main-<date>.log;
- The descriptive rule stats format looks like this:

```
-----rs start-----
Rule statistics for repository <name> :
RULE: ...

Time overall (all rules): ... ns.
-----rs end-----
```

- Stats are printed for each active repository.
- Stats are cumulative, so find the last section `rs start ... rs end` for your repo of interest.
- Rule variants are ordered by total time (descending).

For example, consider the following rule :

```
Id: ptop_PropRestr
  t <ptop:premise>      p
  t <ptop:restriction>  r
  t <ptop:conclusion>   q
  t <rdf:type>          <ptop:PropRestr>
  x p y
  x r y
  -----
  x q y
```

This is a conjunction of two props. It is declared with the axiomatic (A-Box) triples involving `t`. Whenever the premise `p` and restriction `r` hold between two resources, the rule infers the conclusion `q` between the same resources, i.e., `p & r => q`

The corresponding log for variant 4 of this rule may look like the following:

```
RULE ptop_PropRestr_4 invoked 163,475,763 times.
ptop_PropRestr_4:
e b f
a ptop_premise b
a rdf_type ptop_PropRestr
e c f
a ptop_restriction c
a ptop_conclusion d
-----
```

```
e d f
```

```
a ptop_conclusion d invoked 1,456,793 times and took 1,814,710,710 ns.
a rdf_type ptop_PropRestr invoked 7,261,649 times and took 9,409,794,441 ns.
a ptop_restriction c invoked 1,456,793 times and took 1,901,987,589 ns.
e c f invoked 17,897,752 times and took 635,785,943,152 ns.
a ptop_premise b invoked 10,175,697 times and took 9,669,316,036 ns.
Fired 1,456,793 times and took 157,163,249,304 ns.
Inferred 1,456,793 statements.
Time overall: 815,745,001,232 ns.
```

Note: Variable names are renamed due to the compilation to Java bytecode.

Understanding the output:

- The premises are checked in the order given in RULE. (The premise statistics printed after the blank line are not in any particular order.)
- Invoked is the number of times the rule variant or specific premise was checked successfully. Tracing through the rule:
 - ptop_PropRestr_4 checked successfully 163 million times: for each incoming triple, since the lead premise (e b f = x p y) is a free pattern.
 - a ptop_premise b checked successfully 10 million times: for each b=p that has an axiomatic triple involving ptop_premise.
This premise was selected because it has only 1 unbound variable a and it is first in the rule text.
 - a rdf_type ptop_PropRestr checked successfully 7 million times: for each ptop_premise that has type ptop_PropRestr.
This premise was selected because it has 0 unbound variables (after the previous premise binds a).
- The time to check each premise is printed in ns.
- Fired is the number of times all premises matched, so the rule variant was fired.
- Inferred is the number of inferred triples.

It may be greater than fired if there are multiple conclusions.

It may be less than fired since a duplicate triple is not inferred a second time.

- Time overall is the total time that this rule variant took.

Excel format

The log records detailed information about each rule and premise, which is very useful when you are trying to understand which of the rules is too time-consuming. However, it can still be overwhelming because of this level of detail.

Therefore, we have developed the `rule-stats.pl` script that outputs a TSV file such as the following:

rule	ver	tried	time	patts	checks	time	fired	time	triples	speed
ptop_PropChain	4	163475763	776.3	5	117177482	185.3	15547176	590.9	9707142	12505

Parameters:

- rule: the rule ID (name);
- ver: the rule version (variant) or “T” for overall rule totals;

- tried, time: the number of times the rule/variant was tried, the overall time in sec;
- patts: the number of triple patterns (premises) in the rule, not counting the leading premise;
- checks, time: the number of times premises were checked, time in sec;
- fired: the number of times all premises matched, so the rule was fired;
- triples: the number of inferred triples;
- speed: inference speed, triples/sec.

Run the script the following way:

```
perl rule-stats.pl main-2014-07-28.log > main-2014-07-28.xls
```

Investigating performance

The following is an example of using the Excel format to investigate where time is spent during rule execution.

Download the `time-spent-during-rule.xlsx` example file, and use it as a template.

rule	ver	tried	time	pat	checks	time	fired	time	triples	speed
rdfs7_subPropertyOf	T	163476295	1999.4	2	191323783	205.8	27848020	1793.5	24903005	12455.5
ptop_PropChain	T	326951600	1185.1	30	219164757	287.5	22874870	897.5	13813732	11656.6
ptop_PropRestr	T	326951561	1184.3	16	78654527	1023.6	2869625	160.7	1457031	1230.3
rdfs2_domain	T	130143111	533.7	1	49558299	76.5	49558299	457.1	25272752	47357.1
ptop_PropChainRestr	T	490427353	482.7	31	148702055	205.7	5365740	277.1	5365740	11115.8
rdfs3_range	T	163475973	304.1	2	87483795	102.5	87483795	201.6	26317789	86531.8
ptop_TypeRestr	T	163475799	201.4	17	36478117	32.1	1469392	169.3	1469382	7296.6
rdf_type	T	33332899	119.6	2	42522646	52.9	9189831	66.7	6861382	57354.7
ptop_PropChainType2	T	360284446	80.1	31	78128145	79.5	31172	0.6	30074	375.5
owl_inverseOf	T	163475946	0.3	2	64792	0.1	64792	0.3	23746	73592.9
owl_SymPropByInvOf	T	27	0				27	0	0	0
owl_invOfBySymProp	T	27	0				27	0	27	14548.9
ptop_PropChainByPropertyChainAxiom	T	44289	0	9	11109	0	32	0	32	2108.6
ptop_TransPropAsChain	T	7	0				28	0	28	4555
rdfs12_member	T	1	0				1	0	1	10133.1
rdfs13_subClass_Literal	T	9	0				9	0	9	9986.6
rdfs6_subPropertyReflective	T	291	0				291	0	291	17319.9
rdfs8_10_subClassReflective	T	76	0				76	0	76	10138.1
			6090.7						105515097	17323.97

Note: These formulas are dynamic, and they are updated every time you change the filters.

To perform your investigation:

1. Open the results in Excel.
2. Set a filter “ver=T”, first looking at the rules in their entirety instead of rule variants.
3. Sort by “time” (fourth column) in descending order.
4. Check which rules are highlighted in red (those that take significantly long and whose speed is substantially lower than average).
5. Pick a rule (for example, PropRestr).
6. Filter by “rule=PropRestr” and “ver<>T” to see its variants.

rule	ver	tried	time	pat	checks	time	fired	time	triples	speed
ptop_PropRestr	4	163475763	815.7	5	38248684	658.6	1456793	157.2	1456793	1785.8
ptop_PropRestr	5	163475763	368.5	5	40405825	365	1412832	3.6	238	0.6
ptop_PropRestr	2	23	0	3	9	0	0	0	0	0
ptop_PropRestr	0	5	0	1	3	0	0	0	0	0
ptop_PropRestr	1	4	0	2	6	0	0	0	0	0
ptop_PropRestr	3	3	0				0	0	0	0
			1184.2						1457031	1230.393

7. Focus on a variant to investigate the reasons for its poorer time and speed performance.

In this example, the first variant you would want to investigate will be `ptop_PropRestr_5`, as it is spending 30% of the time of this rule, and has very low speed. The reason is that it fired 1.4 million times but produced only 238 triples, so most of the inferred triples were duplicates.

You can find the definition of this variant in the log file:

```

RULE ptop_PropRestr_5 invoked 163,475,763 times.
ptop_PropRestr_5:
e c f
a ptop_restriction c
a rdf_type ptop_PropRestr
e b f
a ptop_premise b
a ptop_conclusion d
-----
e d f

```

It is very similar to the productive variant `ptop_PropRestr_4` (see [Log file](#) above):

- one checks `e b f`. `a ptop_premise b` first,
- the other checks `e c f`. `a ptop_restriction c` first.

Still, the function of these premises in the rule is the same and therefore the variant `ptop_PropRestr_5` (which is checked after 4) is unproductive.

The most likely way to improve performance would be if you make the two premises use the same axiomatic triple `ptop:premise` (emphasizing they have the same role), and introduce a Cut:

```

Id: ptop_PropRestr_SYM
t <ptop:premise>    p
t <ptop:premise>    r
t <ptop:conclusion> q
t <rdf:type>        <ptop:PropRestr>
x p y
x r y               [Cut]
-----
x q y

```

The Cut eliminates the rule variant with `x r y` as leading premise. It is legitimate to do this, since the two variants are the same, up to substitution $p \leftarrow r$.

Note: Introducing a Cut in the original version of the rule would **not** be legitimate:

```

Id: ptop_PropRestr_CUT
t <ptop:premise>    p
t <ptop:restriction> r
t <ptop:conclusion> q
t <rdf:type>        <ptop:PropRestr>
x p y
x r y               [Cut]
-----
x q y

```

since it would omit some potential inferences (in the case above, 238 triples), changing the semantics of the rule (see the example below).

Assume these axiomatic triples:

```

:t_CUT a ptop:PropRestr; ptop:premise :p; ptop:restriction :r; ptop:conclusion :q. # for ptop_
↪PropRestr_CUT
:t_SYM a ptop:PropRestr; ptop:premise :p; ptop:premise      :r; ptop:conclusion :q. # for ptop_
↪PropRestr_SYM

```

Now consider a sequence of inserted triples $x : p : y$, $x : r : y$.

- `ptop_PropRestr_CUT` **will not** infer $x : q : y$, since no variant is fired by the second incoming triple $x : r : y$: it is matched against $x : p : y$, but there is no axiomatic triple $t : \text{ptop:premise} : r$
- `ptop_PropRestr_SYM` **will** infer $x : q : y$, since the second incoming triple $x : r : y$ will match $x : p : y$ and $t : \text{ptop:premise} : r$, then the previously inserted $x : p : y$ will match $t : \text{ptop:premise} : p$ and the rule will fire.

Tip: Rule execution is often non-intuitive, therefore we recommend that you detail the speed history and compare the performance after each change.

Hints on optimizing GraphDB's rulesets

The complexity of the ruleset has a significant effect on the loading performance, the number of inferred statements, and the overall size of the repository after inferencing. The complexity of the standard rulesets increases as follows:

- no inference (lowest complexity, best performance)
- RDFS-Optimized
- RDFS
- RDFS-Plus-Optimized
- RDFS-Plus
- OWL-Horst-Optimized
- OWL-Horst
- OWL-Max-Optimized
- OWL-Max
- OWL2-QL-Optimized
- OWL2-QL
- OWL2-RL-Optimized
- OWL2-RL (highest complexity, worst performance)

It needs to be mentioned that OWL-RL and OWL-QL do a lot of heavy work that is often not required by applications. For more details, see [OWL compliance](#).

Know what you want to infer

Check the expansion ratio (total/explicit statements) for your dataset in order to get an idea of whether this is the result that you are expecting. For example, if your ruleset infers 4 times more statements over a large number of explicit statements, this will take time regardless of the ways in which you try to optimize the rules.

Minimize the number of rules

The number of rules and their complexity affects inferencing performance, even for rules that never infer any new statements. The reason for this is that every incoming statement is passed through every variant of every rule to check whether something can be inferred. This often results in many checks and joins, even if the rule never fires.

So, start with a minimal ruleset and only add the rules that you need. The default ruleset (RDFS-Plus-Optimized) works for many users, but you might even consider starting from RDFS. For example, if you need `owl:Symmetric`

and `owl:inverseOf` on top of RDFS, you can copy only these rules from OWL-Horst to RDFS and leave out the rest.

Conversely, you can start with a bigger standard ruleset and remove the rules that you do not need.

Note: To deploy a custom ruleset, set the *ruleset configuration parameter* to the full pathname of your custom .pie file.

Write your rules carefully

- Be careful with the recursive rules as they can lead to an explosion in the number of inferred statements.
- Always check your spelling:
 - A misspelled variable in a premise leads to a Cartesian explosion (variables quickly growing to an intractable level) of the number of triple joins to be considered by the rule.
 - A misspelled variable in a conclusion (or the use of an unbound variable) leads to the creation of new blank nodes. This is almost never what you really want.
- Order premises by specificity. GraphDB first checks premises with the least number of unbound variables. But if there is a tie, it follows the order given by you. Since you may know the cardinalities of triples in your data, you may be in a better position to determine which premise has better specificity (selectivity).
- Use [Cut] for premises that have the same role (for an example, see *Investigating performance*), but be careful not to remove any necessary inferences by mistake.

Avoid duplicate statements

Avoid inserting explicit statements in a named graph if the same statements are inferable. GraphDB always stores inferred statements in the default graph, so this will lead to duplicating statements. This will increase the repository size and slow down query answering.

You can eliminate duplicates from query results using `DISTINCT` or `FROM onto:skip-redundant-implicit` (see *Other special GraphDB query behaviour*). However, these are slow operations, so it is better not to produce duplicate statements in the first place.

Know the implications of ontology mapping

People often use `owl:equivalentProperty`, `owl:equivalentClass` (and less often `rdfs:subPropertyOf`, `rdfs:subClassOf`) to map ontologies. However, every such assertion means that many more statements are inferred (`owl:equivalentProperty` works as a pair of `rdfs:subPropertyOf`, and `owl:equivalentClass` works as a pair of `rdfs:subClassOf`).

A good example is DCTerms (DCT): almost each DC property has a declared DCT subproperty and there is also a hierarchy amongst DCT properties, for instance:

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

This means that every `dcterms:created` statement will expand to 3 statements. So, do not load the DC ontology unless you really need these inferred `dc:date`.

Consider avoiding inverse statements

Inverse properties (e.g., `:p owl:inverseOf :q`) offer some convenience in querying, but are never necessary:

- SPARQL natively has bidirectional data access: instead of `?x :q ?y`, you can always query for `?y :p ?x`.

- You can even invert the direction in a property path: instead of `?x :p1/:q ?y`, use `?x :p1/(^:p) ?y`.

If an ontology defines inverses but you skip inverse reasoning, you have to check which of the two properties is used in a particular dataset, and write your queries carefully.

The Provenance Ontology (PROV-O) has considered this dilemma thoroughly, and has abstained from defining inverses to “*avoid the need for OWL reasoning, additional code, and larger queries*” (see <http://www.w3.org/TR/prov-o/#inverse-names>).

Consider avoiding long transitive chains

A chain of N transitive relations (e.g., `rdfs:subClassOf`) causes GraphDB to infer and store a further $(n^2 - n)/2$ statements. If the relationship is also symmetric (e.g., in a family ontology with a predicate such as `relatedTo`), then there will be $n^2 - n$ inferred statements.

Consider removing the transitivity and/or symmetry of relations that make long chains. Or, if you must have them, consider the implementation of *TransitiveProperty through step property*, which can be faster than the standard implementation of `owl:TransitiveProperty`.

Consider specialized property constructs

While OWL2 has very powerful class constructs, its property constructs are quite weak. Some widely used OWL2 property constructs could be done faster.

See this [draft](#) for some ideas and clear illustrations. Below, we describe 3 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:

```
Id: ptop_PropChain
  t <ptop:premise1> p1
  t <ptop:premise2> p2
  t <ptop:conclusion> q
  t <rdf:type> <ptop:PropChain>
  x p1 y
  y p2 z
  -----
  x q z
```

It is used with axiomatic triples as in the following:

```
@prefix ptop: <http://www.ontotext.com/proton/protontop#>.
:t a ptop:PropChain; ptop:premise1 :p1; ptop:premise2 :p2; ptop:conclusion :q.
```

transitiveOver

`psys:transitiveOver` has been part of Ontotext’s PROTON ontology since 2008. It is defined as follows:

```
Id: psys_transitiveOver
p <psys:transitiveOver> q
x p y
y q z
-----
x p z
```

It is a specialised PropChain, where premise1 and conclusion coincide. It allows you to chain p with q on the right, yielding p. For example, the inferencing of types along the class hierarchy can be expressed as:

```
rdf:type psys:transitiveOver rdfs:subClassOf
```

TransitiveProperty through step property

owl:TransitiveProperty is widely used and is usually implemented as follows:

```
Id: owl_TransitiveProperty
p <rdf:type> <owl:TransitiveProperty>
x p y
y p z
-----
x p z
```

You may recognize this as a self-chain, thus a specialization of psys:transitiveOver, i.e.:

```
?p rdf:type owl:TransitiveProperty <=> ?p psys:transitiveOver ?p
```

Most transitive properties comprise transitive closure over a basic ‘step’ property. For example, skos:broaderTransitive is based on skos:broader and is implemented as:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive a owl:TransitiveProperty.
```

Now consider a chain of N skos:broader between two nodes. The owl_TransitiveProperty rule has to consider every split of the chain, thus inferring the same closure between the two nodes N times, leading to quadratic inference complexity.

This can be optimized by looking for the step property s and extending the chain only at the right end:

```
Id: TransitiveUsingStep
p <rdf:type> <owl:TransitiveProperty>
s <rdfs:subPropertyOf> p
x p y
y s z
-----
x p z
```

However, this would not make the same inferences as owl_TransitiveProperty if someone inserts the transitive property explicitly (which is a bad practice).

A more robust approach is to declare the step and transitive properties together using psys:transitiveOver, for instance:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive psys:transitiveOver skos:broader.
```

Additional ruleset usage optimization

GraphDB applies the following rules to optimize the ruleset usage so that inferred statements such as `<P a rdf:Property>`, `<P rdfs:subPropertyOf P>` and `<X a rdfs:Resource>` can appear in the repository:

```
/*partialRDFS*/
Id: rdf1_rdfs4a_4b
x a y
-----
a <rdf:type> <rdf:Property>
x <rdf:type> <rdfs:Resource>
a <rdf:type> <rdfs:Resource>
y <rdf:type> <rdfs:Resource>
/*partialRDFS*/

Id: rdfs6
a <rdf:type> <rdf:Property>
-----
a <rdfs:subPropertyOf> a
```

According to them, whatever statement comes into the repository, its subject, predicate and object are resources and its predicate is an `rdf:Property`, which then becomes `subPropertyOf` itself using the second rule (the reflexivity of `subPropertyOf`). These rules, however, if executed every time, present a similar challenge to when using `owl:sameAs`. To avoid the performance drop, GraphDB obtains these statements through code so that `<P a rdf:Property>` and `<X a rdfs:Resource>` are asserted only once – when a property or a resource is encountered for the first time (except in the ‘optimized’ rulesets, where `rdfs:Resource` is omitted because of the very limited use of such inference).

If we start with the empty ruleset, `<P a rdf:Property>`, `<P rdfs:subPropertyOf P>` and `<X a rdfs:Resource>` statements won’t 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.

4.12.3.3 Optimization of `owl:sameAs`

What’s in this document?

- *Removing `owl:sameAs` statements*
- *Disabling the `owl:sameAs` support*
- *How disable-`sameAs` interferes with the different rulesets*
 - *Example 1*
 - *Example 2*
 - *Example 3*
 - *Example 4*
 - *Example 5*

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. $A \rightarrow A$
- Symmetricity, i.e. if $A \rightarrow B$ then $B \rightarrow A$

- Transitivity, i.e. if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

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` behaviour may be beneficial in some cases. For example, it can save you time or you may want to visualize your data without the statements generated by `owl:sameAs` in queries or inferences of such statements.

To disable `owl:sameAs`, use:

- (for individual queries) `FROM onto:disable-sameAs system graph;`
- (for the whole repository) the `disable-sameAs` configuration parameter (Boolean, defaults to 'false'). This disables all inferences.

Disabling owl:sameAs by query does not remove the inferences that have taken place because of owl:sameAs.

Consider the following example:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  <urn:A> owl:sameAs <urn:B> .
  <urn:A> a <urn:Class1> .
  <urn:B> a <urn:Class2> .
}
```

This leads to <urn:A> and <urn:B> being instances of the intersection of the two classes:

```
PREFIX : <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  :Intersection owl:intersectionOf (<urn:Class1> <urn:Class2>) .
}
```

If you query what instances the intersection has:

```
PREFIX : <http://test.com/>

SELECT * {
  ?s a :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.

How disable-sameAs interferes with the different rulesets

The following parameters can affect the owl:sameAs behaviour:

- **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 graphdb-ruleset-usage-optimisation).
- **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 behaviour:

Example 1

If you use owl:sameAs with the following statements:

```
PREFIX : <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  :a :b :c .
  :a owl:sameAs :d .
  :d owl:sameAs :e .
}
```

and you want to retrieve data with this query:

```
PREFIX : <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

DESCRIBE :a :b :c :d :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:

```
:a :b :c .
:a owl:sameAs :a .
:a owl:sameAs :d .
:a owl:sameAs :e .
:b a rdf:Property .
:b rdfs:subPropertyOf :b .
: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 .
```

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 : <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

DESCRIBE :a :b :c :d :e
FROM onto:implicit
FROM onto:disable-sameAs
```

yields:

```
:b a rdf:Property .
:b rdfs:subPropertyOf :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 : <http://test.com/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT DATA {
  GRAPH :graph {
    :a :b :c .
    :a owl:sameAs :d .
    :d owl:sameAs :e .
  }
}
```

Then the test query will be:

```
PREFIX : <http://test.com/>
PREFIX onto: <http://www.ontotext.com/>

SELECT DISTINCT *
{
  GRAPH ?g {
    ?s ?p ?o
    FILTER (
      ?s IN (:a, :b, :c, :d, :e, :graph) ||
      ?p IN (:a, :b, :c, :d, :e, :graph) ||
      ?o IN (:a, :b, :c, :d, :e, :graph) ||
      ?g IN (:a, :b, :c, :d, :e, :graph)
    )
  }
}
```

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

```
graph {
  :a :b :c .
  :a owl:sameAs :d .
  :d owl:sameAs :e .
}
```

because the statements from the default graph are not automatically included. This is the same as in the DESCRIBE query, where using both FROM onto:explicit and FROM onto:implicit nullifies them.

So, if you want to see all the statements, you should write:

```
PREFIX : <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 (:a, :b, :c, :d, :e, :graph) ||
      ?p IN (:a, :b, :c, :d, :e, :graph) ||

```

```

    ?o IN (:a, :b, :c, :d, :e, :graph) ||
    ?g IN (:a, :b, :c, :d, :e, :graph)
  )
}
}
ORDER BY ?g ?s

```

Note that when querying quads, you should use the FROM NAMED clause and when querying triples - FROM. Using FROM NAMED with triples and FROM with quads has no effect and the query will return the following:

```

:graph {
  :a :b :c .
  :a owl:sameAs :d .
  :d owl:sameAs :e .
}
onto:implicit {
  :b a rdf:Property .
  :b rdfs:subPropertyOf :b .
}
onto:implicit {
  :a owl:sameAs :a .
  :a owl:sameAs :d .
  :a owl:sameAs :e .
  :d owl:sameAs :a .
  :d owl:sameAs :d .
  :d owl:sameAs :e .
  :e owl:sameAs :a .
  :e owl:sameAs :d .
  :e owl:sameAs :e .
}
onto:implicit {
  :d :b :c .
  :e :b :c .
}

```

In this case, the explicit statements `<:a owl:sameAs :d>` and `<:d owl:sameAs :e>` appear also as implicit. They do not appear twice when dealing with triples because the iterators return unique triples. When dealing with quads, however, you can see all statements.

Here, you have the same effects with FROM NAMED onto:explicit, FROM NAMED onto:implicit and FROM NAMED onto:disable-sameAs, and the behavior of the `<type Property>`.

4.12.3.4 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 rulesets are provided. These have -optimized appended to the ruleset name, e.g., owl-horst-optimized.

The following optimizations are enacted in GraphDB:

Optimization	Affects patterns
Remove axiomatic triples	• <any> <any> <rdfs:Resource> • <rdfs:Resource> <any> <any> • <any> <rdfs:domain> <rdf:Property> • <any> <rdfs:range> <rdf:Property> • <owl:sameAs> <rdf:type> • <owl:symmetricProperty> • <owl:sameAs> <rdf:type> • <owl:TransitiveProperty>
Remove rule conclusions	• <any> <any> <rdfs:Resource>
Remove rule constraints	• [Constraint <variable> != <rdfs:Resource>]

4.13 Troubleshooting

4.13.1 Database health checks

What's in this document?

- *Possible values for health checks and their meaning*
- *Default health checks for the different GraphDB editions*
- *Aggregated health checks*
 - *Running health checks*
- *Running legacy health checks*

The GraphDB health check endpoint is at <http://localhost:7200/repositories/myrepo/health>.

Possible responses: HTTP status 200 (the repository is healthy), 206 (the repository needs attention but it is not something critical), 500 (the repository is inconsistent, i.e. some checks failed).

4.13.1.1 Possible values for health checks and their meaning

Value	Description
read-availability	Checks whether the repository is readable.
storage-folder	Checks if there are at least 20 megabytes writable left for the storage folder. The amount can be controlled with the system parameter <code>health.minimal.free.storage</code> .
long-running-queries	Checks if there are queries running longer than 20 seconds. The time can be controlled with the system parameter <code>health.max.query.time.seconds</code> . If a query is running for more than 20 seconds, it is either a slow one, or there is a problem with the database.
predicates-statistics	Checks if the predicate statistics contain correct values.
master-status	Checks whether the master is up and running, can access its workers, and the peers are not lagging. If there are non-readable workers, the status will be yellow. If there are workers that are off, the status will be red.
plugins	Provides aggregated health checks for the individual plugins.

4.13.1.2 Default health checks for the different GraphDB editions

Name	Free	SE	EE / Worker	EE / Master
read-availability	✓	✓	✓	✓
storage-folder	✓	✓	✓	✓
long-running-queries	✓	✓	✓	×
predicates-statistics	✓	✓	✓	×
master-status	×	×	×	✓
plugins	✓	✓	✓	×

4.13.1.3 Aggregated health checks

The aggregated GraphDB health checks include checks for dependent services and components as plugins and connectors.

Each connector plugin is reported independently as part of the composite “plugins” check in the repository health check. Each connector’s check is also a composite where each component is an individual connector instance.

The output may look like this:

```
{
  "name": "wine",
  "status": "green",
  "components": [
    {
      "name": "read-availability",
      "status": "green"
    },
    {
      "name": "storage-folder",
      "status": "green"
    },
    {
      "name": "long-running-queries",
      "status": "green"
    },
    {
      "name": "predicates-statistics",
      "status": "green"
    },
    {
      "name": "plugins",
      "status": "yellow",

```

```
"components":[
  {
    "name":"elasticsearch-connector",
    "status":"green",
    "components":[

    ]
  },
  {
    "name":"lucene-connector",
    "status":"yellow",
    "components":[
      {
        "name":"my_index",
        "status":"green",
        "message":"query took 0 ms, 5 hits"
      },
      {
        "name":"my_index2",
        "status":"yellow",
        "message":"query took 0 ms, 0 hits"
      }
    ]
  },
  {
    "name":"solr-connector",
    "status":"yellow",
    "components":[
      {
        "name":"my_index",
        "status":"green",
        "message":"query took 7 ms, 5 hits"
      },
      {
        "name":"my_index2",
        "status":"yellow",
        "message":"query took 5 ms, 0 hits"
      }
    ]
  }
]
}
]
```

An individual check run involves sending a query for all documents to the connector instance, and the result is:

- green - more than zero hits
- yellow - zero hits or failing shards (shards check only for Elasticsearch)
- 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 command:

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


4.13.1.4 Running legacy health checks

If you have been relying on the health check format from a version of GraphDB prior to 9.0, you can add the old parameter to get results in the old health check format:

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

Parameter: checks (By default all checks are run)

Behavior: Run only the specified checks.

Accepts multiple values: True.

Values: read-availability, storage-folder, long-running-queries, predicates-statistics, master-status.

```
curl 'http://localhost:7200/repositories/myrepo/health?old&checks=<value1>&checks=<value2>'
```

- an example output for a healthy repository with HTTP status 200:

```
{
  "predicates-statistics": "OK",
  "long-running-queries": "OK",
  "read-availability": "OK",
  "status": "green",
  "storage-folder": "OK"
}
```

- an example output for an unhealthy repository with HTTP status 500:

```
{
  predicates-statistics: "OK",
  long-running-queries: "OK",
  read-availability: "OK",
  storage-folder: "UNHEALTHY: Permission denied java.io.IOException: Permission denied",
  status: "red"
}
```

The status field in the output means the following:

- green - all is good;
- yellow - the repository needs attention;
- red - the repository is inconsistent in some way.

4.13.2 System metrics monitoring

What's in this document?

- *Page cache metrics*
- *Entity pool metrics*

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>`.

Attributes		Operations	Notifications	Metadata
Attribute values				
	Name			Value
	50thPercentile			0.292526
	75thPercentile			0.39041499999999996
	95thPercentile			0.439266
	98thPercentile			0.711468
	999thPercentile			0.8529749999999999
	99thPercentile			0.8529749999999999
	Count			28
	DurationUnit			milliseconds
	FifteenMinuteRate			0.26024847734263823
	FiveMinuteRate			0.0019648345076526827
	Max			0.8529749999999999
	Mean			0.2686243969344068
	MeanRate			0.012647597864901165
	Min			0.003972
	OneMinuteRate			4.3066832839259026E-16
	RateUnit			events/second
	StdDev			0.1782919456029322

4.13.2.1 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:

- `cache.flush` - 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.
- `cache.hit` - number of hits in the cache. This can be viewed as the number of pages that do not need to be read from the disc but can be taken from the cache.
- `cache.load` - a **timer** for the pages that have to be read from the disc. The smaller the number of pages is, the better.
- `cache.miss` - number of cache misses. The smaller this number is, the better. If you see that the number of hits is smaller than the misses, then it is probably a good idea to increase the page cache memory.

4.13.2.2 Entity pool metrics

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

- `epool.read` - a **timer** for the number of reads in the entity pool.
- `epool.write` - a **timer** for the number of writes in the entity pool.

4.13.3 Diagnosing and reporting critical errors

What's in this document?

- *Report*
 - *Report content*
 - *Create Report from Workbench*
 - *Create report through the report script*
- *Logs*
 - *Setting up the root logger*
 - *Logs location*
 - *Log files*

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.

4.13.3.1 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 your distribution. The report is saved in *GraphDB-Work*/report directory. There is always one report - the one that has been generated most recently.

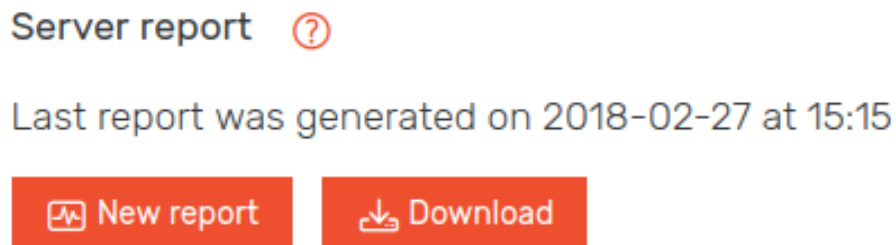
Report content

- 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* as `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 `java GC.class_histogram` as `gcmd_histogram.txt`
- the output from `java Thread.print` as `thread_dump.txt`
- the System Properties for the GraphDB instance
- the repository configurations info as `system.ttl`
- the `owlim.properties` file for each repository

Archive: graphdb-server-report-2018-02-28_163747+0200.zip			
Length	Date	Time	Name
12	2018-02-28	16:37	gdb_version.txt
41514	2018-02-28	16:37	home.txt
31203	2018-02-28	16:37	work.txt
9275	2018-02-28	16:37	data.txt
0	2018-02-28	16:37	logs/
1234	2018-02-28	16:37	logs/main-2017-08-30.log
0	2018-02-28	16:37	logs/error-2017-08-28.log
0	2018-02-28	16:37	logs/error-2017-08-30.log
21210	2018-02-28	16:37	logs/main-2017-08-28.log
0	2018-02-28	16:37	logs/query-log-2017-08-30.log
19047	2018-02-28	16:37	logs/query-log-2017-08-28.log
0	2018-02-28	16:37	conf/
0	2018-02-28	16:37	repositories/
0	2018-02-28	16:37	repositories/repo/
734	2018-02-28	16:37	repositories/repo/owlim.properties
0	2018-02-28	16:37	workbench/
48614	2018-02-28	16:37	workbench/settings.js
485345	2018-02-28	16:37	jcmd_histogram.txt
29762	2018-02-28	16:37	thread_dump.txt
4454	2018-02-28	16:37	system.properties
2131	2018-02-28	16:37	system.ttl
694535	21 files		

Create Report from Workbench

Go to *Help -> System information*. Click on *Create new server report* in the *Application info* tab to obtain a new one, wait until it is ready, and download it.



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`.

4.13.3.2 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 ] 2016-03-17 17:29:31,657 [main | ROOT] Using 'file:/opt/graphdb-ee/conf/logback.xml' as
↳ logback's configuration file for graphdb
```

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 don't have to restart the database as it will check the file for changes every 30 seconds, and will reconfigure the logger.

- Change the log level through the `logback JMX` configurator. For more information, see the [Logback manual chapter 10](#).
- Start each component with `graphdb.logger.root.level` set to your desired root logging level. For example:

```
bin/graphdb -Dgraphdb.logger.root.level=WARN
```

Logs location

By default, all database components and tools log in *GraphDB-HOME/logs*, when run from the bin folder. If you setup GraphDB by deploying .war files into a stand-alone 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.

- *http-log.log* - contains the HTTP communication between the master and the workers.
- *query-log.log* - contains all queries that were sent to the database. The format is machine-readable and allows us to replay the queries when debugging a problem.
- *main.log* - contains all messages coming from the main part of the engine.

4.13.4 Storage tool

What's in this document?

- *Options*
- *Supported commands*
- *Examples*

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 ./storage-tool help.

Note: The tool works only on repository images that are not in use (i.e., when the database is down).

4.13.4.1 Options

```
-command=<operation to be executed, MANDATORY>
-storage=<absolute path to repo storage dir, MANDATORY>
-eseize=<size of entity pool IDs: 32 or 40 bits, DEFAULT 32>
-statusPrintInterval=<size of the external sort buffer, DEFAULT 95, means 95M elements, max value is 100, also 95>
-pageCacheSize=<size of the page cache, DEFAULT 10, means 10K elements>
-sortBufferSize=<size of the external sort buffer, DEFAULT 100, means 100M elements>
-srcIndex=<one of pso, pos>
-destIndex=<one of pso, pos, cpso>
-origURI=<original existing URI in the repo>
-replURI=<new non-existing URI in the repo>
-destFile=<path to file used to store exported data>
```

4.13.4.2 Supported commands

- `scan` - scans the repository index(es) and prints statistics about the number of statements and repository consistency;
- `rebuild` - 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`;
- `replace` - replaces an existing entity `-origURI` with a non-existing one `-replURI`;
- `repair` - repairs the repository indexes and restores data, a better variant of the merge index;
- `export` - uses the source index (`srcIndex`) to export repository data to the destination file `destFile`. Supported destination file extensions formats: `.trig`, `.ttl`, `.nq`;
- `epool` - scans the entity pool for inconsistencies and checks for invalid IRIs. IRIs are validated against the [RFC 3987](#) standard. Invalid IRIs will be listed in an `entities.invalid.log` file for review. If `-fix` is specified, instead of listing the invalid IRIs, they will instead be fixed in the entity pool.

4.13.4.3 Examples

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

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

– when everything is OK

```
Scan result consistency check.

-----scan results-----
mask |          pso |          pos | diff | flags
0001 |         21,166 |         21,166 | OK | INF
0002 |        151,467 |        151,467 | OK | EXP
0005 |             70 |             70 | OK | INF RO

-----additional checks-----
|          pso |          pos | stat | check-type
|        77cdab99 |        77cdab99 | OK | checksum
|             0 |             0 | OK | not existing ids
|             0 |             0 | OK | literals as subjects
|             0 |             0 | OK | literals as predicates
|             0 |             0 | OK | literals as contexts
|             0 |             0 | OK | blanks as predicates
|           true |           true | OK | page consistency
|        9c564c22 |        9c564c22 | OK | cpso crc
|             - |             - | OK | epool duplicate ids
|             - |             - | OK | epool consistency
|             - |             - | OK | literal index consistency

Scan determines that this repo image is consistent.
```

– when there are broken indexes

```
-----scan results-----
mask |          pso |          pos | diff | flags
0001 |    29,284,580 |    29,284,580 | OK | INF
0002 |    63,559,252 |    63,559,252 | OK | EXP
0004 |         8,134 |         8,134 | OK | RO
0005 |         1,140 |         1,140 | OK | INF RO
```

0009		1,617,004		1,617,004		OK		INF	HID
000a		3,068,289		3,068,289		OK		EXP	HID
0011		1,599,375		1,599,375		OK		INF	EQ
0012		2,167,536		2,167,536		OK		EXP	EQ
0020		327		254		OK		DEL	
0021		11		12		OK		INF	DEL
0022		31		24		OK		EXP	DEL
004a		17		17		OK		EXP	HID MRK

```

-----additional checks-----
|          pso |          pos | stat | check-type
| ffffffff93e6a372 | ffffffff93e6a372 | OK | checksum
|          0 |          0 | OK | not existing ids
|          0 |          0 | OK | literals as subjects
|          0 |          0 | OK | literals as predicates
|          0 |          0 | OK | literals as contexts
|          0 |          0 | OK | blanks as predicates
|          true |          true | OK | page consistency
| bf55ab00 | bf55ab00 | OK | cpso crc
|          - |          - | OK | epool duplicate ids
|          - |          - | OK | epool consistency
|          - |          - | ERR | literal index consistency

```

Scan determines that this repo image is INCONSISTENT.

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:

```
-command=scan -storage=/repo/storage -srcIndex=pso -esize=40 -statusPrintInterval=60
```

- compact the PSO index (self-rebuild equals compacting):

```
-command=rebuild -storage=/repo/storage -esize=40 -srcIndex=pso -destIndex=pso
```

- rebuild the POS index from the PSO index and compact POS:

```
-command=rebuild -storage=/repo/storage -esize=40 -srcIndex=pso -destIndex=pos
```

- rebuild the predicates statistics index:

```
-command=rebuild -storage=/repo/storage -esize=40 -destIndex=predicates
```

- replace `http://onto.com#e1` with `http://onto.com#e2`:

```
-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:

```
-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:

```
-command=epool -storage=/repo/storage -esize=40
```


5.1 Loading Data

GraphDB exposes multiple interfaces for loading RDF data. It also supports the conversion of tabular data into RDF and its direct load into an active repository, using simple SPARQL queries and a virtual endpoint. This functionality is based on [OpenRefine](#), and the supported formats are TSV, CSV, Excel (.xls and .xlsx), JSON, XML or Google sheet.

Table 5.1: GraphDB's data loading interfaces.

Interface	Use cases	Mode	Speed
SPARQL endpoint	No limits on the file size	Online parallel	Moderate speed
Workbench import a local or a remote RDF file	Small files limited up to 200MB	Online parallel	Moderate speed
Workbench import a server file	No limits on the file size	Online parallel	Fast ignoring all HTTP protocol overheads
LoadRDF	Batch import of very big files	Initial offline import with no plugins	Fast with a small speed degradation
Preload	Import huge datasets with no inference	Initial offline import with no inference and plugins	Ultra fast without speed degradation
OntoRefine	Import and clean non RDF based formats	In-memory operation limited to the available heap	Slow

5.1.1 Loading data using the Workbench

What's in this document?

- *Import settings*
- *Importing local files*
- *Importing remote content*
- *Importing RDF data from a text snippet*
- *Importing server files*
- *Import data with an INSERT query*

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

In case that you have many files, a file name filter is available to narrow down the list.

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 BNode 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.

Import settings

×

Base IRI ⓘ

http://exampleuri.com/examplepath

Target graph ⓘ

☒ From data
 ☐ The default graph
 ☐ Named graph

http://example.com/graph...

☐ Enable replacement of existing data

Show advanced settings ▾

Restore defaults

Cancel

Import

5.1.1.2 Importing local files

Go to *Import* -> *RDF* -> *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 icon 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.

Import ?

User data

Server files

Help

Upload RDF files
All RDF formats, up to 200 MB

Get RDF data from a URL
All RDF formats

Import RDF text snippet
Type or paste RDF data

☐

Import

Reset status

Remove

Type to filter

☒	graphdb-news-dataset.nt	Imported successfully in less than a second.	Import
☒	pub-ontology-types.ttl	Imported successfully in less than a second.	Import
☒	pub-ontology.ttl	Imported successfully in less than a second.	Import
☒	pub-properties.ttl	Imported successfully in less than a second.	Import
☒	publishing-ontology.ttl	Imported successfully in less than a second.	Import

5.1.1.3 Importing remote content

Go to *Import* -> *RDF* -> *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.

Import RDF data from URL

×

https://www.w3.org/TR/owl-guide/wine.rdf

☒ Start import automatically

Cancel

Format: Auto

Import

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* -> *RDF* -> *User data* -> *Import RDF text snippet*.

You can import data by typing or pasting it directly in the *Text area* control. This very simple text import sends the data to the [Repository Statements Endpoint](#).

Import RDF data from a text snippet



```
# Example: rdf:predicate a rdf:Property .
```

☒ Start import automatically

Cancel

Format: Turtle ▾

Import

5.1.1.5 Importing server files

Go to *Import -> RDF -> 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.

SPARQL Query & Update

Editor only Editor and results Results only

```

1 PREFIX dc: <http://purl.org/dc/elements/1.1/>
2 INSERT DATA
3 {
4   GRAPH <http://example> {
5     <http://example/book1> dc:title "A new book" ;
6                           dc:creator "A.N.Other" .
7   }
8 }

```







5.1.2 Loading data using the LoadRDF tool

What's in this document?

- *Command Line Options*
 - *Load data in a repository created from the Workbench*
 - *Load data in a new repository initialized by a config file*

- [A GraphDB Repository Configuration Sample](#)
- [Tuning LoadRDF](#)

LoadRDF is a tool designed for offline loading of data sets. 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 indices and producing a ready-to-use repository.

The LoadRDF 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.

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: For loading datasets bigger than several billion RDF statements, consider using the [Preload tool](#).

5.1.2.1 Command Line Options

```
usage: loadrdf [OPTION]... [FILE]...
Loads data in a newly created repository or overwrites an existing one.
-c,--configFile <file_path>    repo definition .ttl file
-f,--force                      overwrite existing repo
-i,--id <repository-id>        existing repository id
-m,--mode <serial|parallel>    singlethread | multithread parse/load/infer
-p,--partialLoad               allow partial load of file that contains
                               corrupt line
-s,--stopOnFirstError          stop process if the dataset contains a
                               corrupt file
-v,--verbose                   print metrics during load
```

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, optionally 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.

Note: The LoadRDF Tool supports `.zip` and `.gz` files, and directories. If specified, the directories can be processed recursively.

There are two common cases for loading data with the LoadRDF tool:

Load data in a repository created from the Workbench

1. Configure LoadRDF repositories location by setting the property `graphdb.home.data` in `<graphdb_dist>/conf/graphdb.properties`. If no property is set, the default repositories location will be: `<graphdb_dist>/data`.
2. Start GraphDB.
3. Start a browser and go to the Workbench Web application using a URL of this form: <http://localhost:7200>. Substituting `localhost` and the `7200` port number as appropriate.
4. Set up [a valid license](#) for the GraphDB.

5. Go to *Setup-> Repositories*.
6. *Create* and *configure* a repository.
7. Shut down GraphDB.
8. Start the bulk load with following command:

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

9. Start GraphDB.

Load data in a new repository initialized by a config file

1. Stop GraphDB.
2. Configure LoadRDF repositories location by setting the property `graphdb.home.data` in `<graphdb-dist>/conf/graphdb.properties`. If no property is set, the default repositories location will be: `<graphdb-dist>/data`.
3. Create a configuration file.
4. Make sure that a *valid license has been configured for the LoadRDF tool*.
5. Start the bulk load with following command:

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

6. Start GraphDB.

5.1.2.2 A GraphDB Repository Configuration Sample

Example configuration template, using minimal parameters set. However, you can add more optional parameters from the `configs/templates` example:

```
#
# Configuration template for an GraphDB-SE 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 owl: <http://www.ontotext.com/trree/owlim#>.

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

      # ruleset to use
      owl:ruleset "rdfsplus-optimized" ;

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

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

```
]
].
```

5.1.2.3 Tuning LoadRDF

The LoadRDF tool accepts java command line options, using `-D`. To change them, edit the command line script.

The following options can tune the behaviour of the parallel loading:

- `-Dpool.buffer.size` - 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 to wait for the operations on which they rely, and the CPU is intensively used most of the time.
- `-Dinfer.pool.size` - the number of inference threads in parallel mode. The default value is the number of cores of the machine processor or 4, as set in the command line scripts. 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.

5.1.3 Loading data using the Preload tool

What's in this document?

- *Preload vs LoadRDF*
- *Command line option*
- *A GraphDB repository configuration sample*
- *Tuning Preload*
- *Resuming data loading with Preload*

Preload is a tool for converting RDF files into GraphDB indices into a very low level. A common use case is the initial load of datasets larger than several billion RDF statements with no inference. Preload can perform only an initial load, which is transactional, supports stop requests, resume and consistent output even after failure. On a standard server with NVMe drive or fast SSD disks, it can sustain a data loading speed of over 130,000 RDF statements per second with no speed degradation.

5.1.3.1 Preload vs LoadRDF

Despite of the many similarities between LoadRDF and Preload, such as the fact that both tools do parallel offline transformation of RDF files into GraphDB image, there are also substantial differences in their implementation. LoadRDF 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.

The Preload tool 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 tool requires almost twice as much disk space to complete the import.

5.1.3.2 Command line option

```
usage: PreloadData [OPTION]... [FILE]...
  Loads data in newly created repository or overwrites existing one.
  -a,--iter.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,--configFile <file_path> repo definition .ttl file
  -f,--force                 overwrite existing repo
  -i,--id <repository-id>    existing repository id
  -p,--partialLoad           allow partial load of file that contains corrupt line
  -q,--queue.folder <arg>    where to store temporary data
  -r,--recursive             walk folders recursively
  -s,--stopOnFirstError      stop process if the dataset contains a corrupt file
  -t,--parsing.tasks <arg>   number of rdf parsers
  -x,--restart               restart load, ignoring an existing recovery point
  -y,--interval <arg>       recover point interval in seconds
```

There are two common cases for loading data with the Preload tool:

Loading data in a repository created from the Workbench

1. Configure Preload repositories location by setting the property `graphdb.home.data` in `<graphdb_dist>/conf/graphdb.properties`. If no property is set, the default repositories location will be: `<graphdb_dist>/data`.
2. Start GraphDB.
3. Start a browser and 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. Set up *a valid license* for GraphDB.
5. Go to *Setup->Repositories*.
6. *Create* and *configure* a repository.
7. Shut down GraphDB.
8. Start the bulk load with the following command:

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

9. Start GraphDB.

Loading data in a new repository initialized by a config file

1. Stop GraphDB.
2. Configure Preload repositories location by setting the property `graphdb.home.data` in `<graphdb_dist>/conf/graphdb.properties`. If no property is set, the default repositories location will be: `<graphdb_dist>/data`.
3. Create a configuration file.
4. Make sure that a *valid license has been configured for the Preload tool*.
5. Start the bulk load with the following command:

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

6. Start GraphDB.

5.1.3.3 A GraphDB repository configuration sample

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

```
#
# Configuration template for a GraphDB-SE 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 owl: <http://www.ontotext.com/trree/owlim#>.

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

      # ruleset to use
      owl:ruleset "empty" ;

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

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

5.1.3.4 Tuning Preload

The Preload tool accepts command line 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.
- `iter.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.

5.1.3.5 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 3600s. (60min.)

- `restart` - if set to `true`, the loading will start from the beginning, ignoring an existing recovery point. The default is `false`.

5.1.4 Loading data using OntoRefine

What's in this document?

- *OntoRefine - overview and features*
- *Example Data*
- *Upload data in OntoRefine*
 - *Open OntoRefine in the Workbench*
 - *Create a project*
 - *Import a project*
 - *Open a project*
- *Viewing tabular data as RDF*
- *RDFizing data*
 - *Using CONSTRUCT query in the OntoRefine SPARQL endpoint*
- *Importing data in GraphDB*
- *Rows and records*
 - *Records*
- *Additional resources*

5.1.4.1 OntoRefine - overview and features

GraphDB OntoRefine is a data transformation tool, based on [OpenRefine](#) and integrated in the GraphDB Workbench. It can be used for converting tabular data into RDF and importing it into a GraphDB repository, using simple SPARQL queries and a virtual endpoint. The supported formats are TSV, CSV, *SV, XLS, XLSX, JSON, XML, RDF as XML, and Google sheet. Using OntoRefine, you can easily filter your data, edit its inconsistencies, convert it into RDF, and import it into a repository.

OntoRefine enables you to:

- Upload your data file(s) and create a project.
- View the cleaned data as RDF.
- Transform your data using [SPIN functions](#).
- Import the newly created RDF directly in a GraphDB repository by using the GraphDB SPARQL endpoint.

5.1.4.2 Example Data

The data used for the examples can be found at:

- `USPresident-Wikipedia-URLs-Thmbs-HS.csv`
- `movie_metadata.csv`

5.1.4.3 Upload data in OntoRefine

Open OntoRefine in the Workbench

To transform your data into RDF, you need a working GraphDB database.

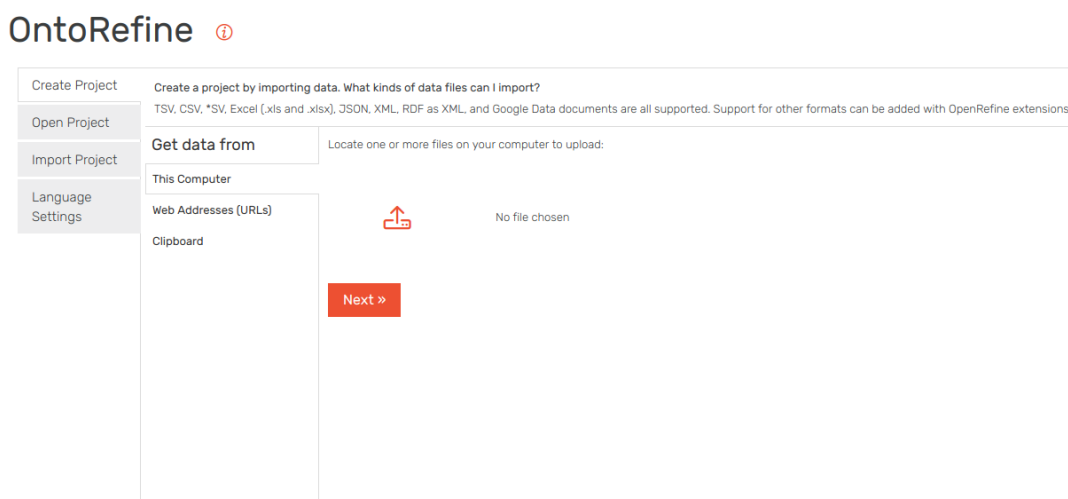
1. Start GraphDB in Workbench mode.
2. Open <http://localhost:7200/> in a browser.
3. Go to *Import -> Tabular (OntoRefine)*.

All data files in OntoRefine are organized as projects. One project can have more than one data file.

The *Create Project* action area consists of three tabs corresponding to the source of data. You can upload a file from your computer, specify the URL of a publicly accessible data, or paste data from the clipboard.

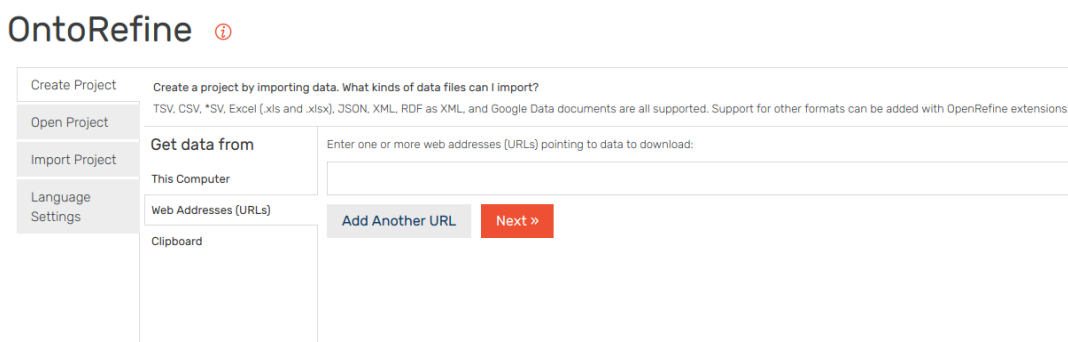
Create a project

1. Click *Create Project -> Get data from*.
2. Select one or more files to upload:
 - from your computer



The screenshot shows the OntoRefine web interface. On the left is a sidebar with a menu: 'Create Project' (highlighted), 'Open Project', 'Import Project', 'Language', and 'Settings'. The main area is titled 'Create a project by importing data. What kinds of data files can I import?' and lists supported formats: TSV, CSV, *SV, Excel (.xls and .xlsx), JSON, XML, RDF as XML, and Google Data documents. Below this is the 'Get data from' section with three tabs: 'This Computer', 'Web Addresses (URLs)', and 'Clipboard'. The 'This Computer' tab is active, showing a file upload icon and the text 'No file chosen'. A red 'Next >' button is at the bottom right.

- from web addresses (URLs)



This screenshot shows the same OntoRefine interface, but the 'Web Addresses (URLs)' tab is now selected. It prompts the user to 'Enter one or more web addresses (URLs) pointing to data to download:' with an empty text input field. Below the input field are two buttons: 'Add Another URL' and a red 'Next >' button. The sidebar and top header remain the same.

- from clipboard

OntoRefine

Create Project

Open Project

Import Project

Language Settings

Create a project by importing data. What kinds of data files can I import?

TSV, CSV, *SV, Excel (.xls and .xlsx), JSON, XML, RDF as XML, and Google Data documents are all supported. Support for other formats can be added with OpenRefine extensions.

Get data from

This Computer

Web Addresses (URLs)

Clipboard

Paste data from clipboard here:

Next »

- Click *Next*.
- (Optional) Change the table configurations and update the preview.

With the first opening of the file, OntoRefine tries to recognize the encoding of the text file and all delimiters.

OntoRefine

Create Project

Open Project

Import Project

Language Settings

« Start Over

USPresident Wikipedia URLs Thmbs + **Create Project »**

Presidency	President	Wikipedia Entry	Took office	Left office	Party	Portrait	Thumbnail	Home State
1	George Washington	http://en.wikipedia.org/wiki/George_Washington	30/04/1789	4/03/1797	Independent	GeorgeWashington.jpg	thmb_GeorgeWashington.jpg	Virginia
2	John Adams	http://en.wikipedia.org/wiki/John_Adams	4/03/1797	4/03/1801	Federalist	JohnAdams.jpg	thmb_JohnAdams.jpg	Massachusetts
3	Thomas Jefferson	http://en.wikipedia.org/wiki/Thomas_Jefferson	4/03/1801	4/03/1809	Democratic-Republican	ThomasJefferson.gif	thmb_ThomasJefferson.gif	Virginia
4	James Madison	http://en.wikipedia.org/wiki/James_Madison	4/03/1809	4/03/1817	Democratic-Republican	JamesMadison.gif	thmb_JamesMadison.gif	Virginia
5	James Monroe	http://en.wikipedia.org/wiki/James_Monroe	4/03/1817	4/03/1825	Democratic-Republican	JamesMonroe.gif	thmb_JamesMonroe.gif	Virginia
6	John Quincy Adams	http://en.wikipedia.org/wiki/John_Quincy_Adams	4/03/1825	4/03/1829	Democratic-Republican/National Republican	JohnQuincyAdams.gif	thmb_JohnQuincyAdams.gif	Massachusetts
7	Andrew Jackson	http://en.wikipedia.org/wiki/Andrew_Jackson	4/03/1829	4/03/1837	Democratic	Andrew_Jackson_head.gif	thmb_Andrew_Jackson_head.gif	Tennessee
8	Martin Van Buren	http://en.wikipedia.org/wiki/Martin_Van_Buren	4/03/1837	4/03/1841	Democratic	MartinVanBuren.gif	thmb_MartinVanBuren.gif	New York
9	William Henry Harrison	http://en.wikipedia.org/wiki/William_Henry_Harrison	4/03/1841	4/04/1841	Whig	WilliamHenryHarrison.gif	thmb_WilliamHenryHarrison.gif	Ohio
10	John Tyler	http://en.wikipedia.org/wiki/John_Tyler	4/04/1841	4/03/1845	Whig	JohnTyler.jpg	thmb_JohnTyler.jpg	Virginia
11	James K. Polk	http://en.wikipedia.org/wiki/James_K_Polk	4/03/1845	4/03/1849	Democratic	JamesKPolk.gif	thmb_JamesKPolk.gif	Tennessee
12	Zachary Taylor	http://en.wikipedia.org/wiki/Zachary_Taylor	4/03/1849	9/07/1850	Whig	ZacharyTaylor.jpg	thmb_ZacharyTaylor.jpg	Louisiana
13	Millard Fillmore	http://en.wikipedia.org/wiki/Millard_Fillmore	9/07/1850	4/03/1853	Whig	MillardFillmore.png	thmb_MillardFillmore.png	New York
								New

Parse data as

Character encoding

Update Preview

Parse data as

CSV / TSV / separator-based files

Line-based text files

Fixed-width field text files

JSON files

MARC files

RDF/N3 files

Wikitext

Columns are separated by

Commas (CSV)

Tabs (TSV)

Custom

Escape special characters with \

Ignore first 0 line(s) at beginning of file

Parse next 1 line(s) as column headers

Discard initial 0 row(s) of data

Load at most 0 row(s) of data

Parse cell text into

numbers, dates, ...

Quotation marks are used to enclose cells containing

column separators

Store blank rows

Store blank cells as nulls

- Click *Create Project*.

The result is a table similar to that of an Excel or a Google sheet.

Import a project

To import an already existing OntoRefine project:

- Go to *Import Project*.

5.1. Loading Data

125

2. Select a file (.tar or .tar.gz)
3. Import it.

OntoRefine

Create Project
Open Project
Import Project
Language Settings

Locate an existing Refine project file (.tar or .tar.gz):

Project file:  No file chosen

Re-name project (optional):

Import Project

Open a project

Once the project is created:

1. Go to *Open Project*.
2. Click the one you want to work on.
3. (Optional) You can also delete your project if you want to.

OntoRefine

Create Project								
Open Project								
Import Project								
Language Settings								
			↕ Last modified	↕ Name	↕ Creator	↕ Subject	↕ Description	↕ Row Count
			2018-06-04 18:32 PM	movie_metadata.csv				5043
			2018-06-01 18:31 PM	USPresident Wikipedia URLs Thmbs HS.csv				44

5.1.4.4 Viewing tabular data as RDF

Unlike OpenRefine that supports RDF for an input, OntoRefine also supports RDF as an output. To view your data in triples, click the OntoRefine *RDF* button. A settings screen appears to configure the RDF data that OntoRefine helps you generate.

RDF settings

Base IRI:

http://example.com/resource/

Base IRI prefix:

mydata

☐

Generate row and record IRIs based on column values

☒

Automatically escape column values for IRIs

☒

Automatically detect IRIs in cell values

☒

Replace cell values with reconciliation IRIs

Row type IRI:

Row

Row number IRI:

rowNumber

Record type IRI:

Record

Record number IRI:

recordNumber

Record id IRI:

recordId

Record to row link IRI:

hasRow

☐

Ignore records

Restore defaults

OK

Cancel

You can change the IRIs generated for your data. Leaving the defaults will return the following query; click *Run* to obtain the results.

OntoRefine
name: USPresident Wikipedia URLs Thmbs HS csv
Open... Data Refine Help

```

3
4 # Example query returning RDF data
5 SELECT * {
6   # Triple patterns for accessing each row and the columns in contains
7   # Note that no triples will be generated for NULL values in the table
8   # You should inspect your data in Refine mode and add OPTIONAL accordingly
9   ?row a mydata:Row ;
10    mydata:Presidency ?Presidency ;
11    mydata:President ?President ;
12    mydata:Wikipedia_Entry ?Wikipedia_Entry ;
13    mydata:Took_office ?Took_office ;
14    mydata:Left_office ?Left_office ;
15    mydata:Party ?Party ;
16    mydata:Portrait ?Portrait ;
17    mydata:Thumbnail ?Thumbnail ;
18    mydata:Home_State ?Home_State .

```

Table Raw Response Pivot Table Google Chart

Filter query results Showing 1 to 44 of 44 entries

	row	Presidency	President	Wikipedia_Entry	Took_office	Left_office	Party	Portrait	Thumbnail	Home_State
1	mydata:Row1	1	George Washington	http://en.wikipedia.org/wiki/George_Washington	30/04/1789	4/03/1797	Independent	GeorgeWashington.jpg	thmb_GeorgeWashington.jpg	Virginia
2	mydata:Row2	2	John Adams	http://en.wikipedia.org/wiki/John_Adams	4/03/1797	4/03/1801	Federalist	JohnAdams.jpg	thmb_JohnAdams.jpg	Massachusetts
3	mydata:Row3	3	Thomas Jefferson	http://en.wikipedia.org/wiki/Thomas_Jefferson	4/03/1801	4/03/1809	Democratic-Republican	Thomasjefferson.gif	thmb_Thomasjefferson.gif	Virginia
4	mydata:Row4	4	James Madison	http://en.wikipedia.org/wiki/James_Madison	4/03/1809	4/03/1817	Democratic-Republican	JamesMadison.gif	thmb_JamesMadison.gif	Virginia
5	mydata:Row5	5	James Monroe	http://en.wikipedia.org/wiki/James_Monroe	4/03/1817	4/03/1825	Democratic-Republican	JamesMonroe.gif	thmb_JamesMonroe.gif	Virginia

Warning: The OntoRefine SPARQL button sends queries only to the currently open OntoRefine project. It should not be mistaken with the GraphDB Workbench *SPARQL* tab, which is directly connected to the current repository.

5.1.4.5 RDFizing data

You can transform your data in the OntoRefine SPARQL endpoint using a *CONSTRUCT* query.

Using CONSTRUCT query in the OntoRefine SPARQL endpoint

Construct new RDF data based on the RDFized model of your tabular data. Use *SPIN* functions to construct proper triples.

GraphDB 8.0 supports *SPIN* functions:

- SPARQL functions for splitting a string
- SPARQL functions for parsing dates
- SPARQL functions for encoding URIs

Parsing dates and encoding URIs

```

PREFIX mydata: <http://example.com/resource/>
PREFIX spif: <http://spinrdf.org/spif#>

# Example query returning RDF data
CONSTRUCT {
  ?presidentIRI a mydata:President ;
  mydata:tookOffice ?tookOfficeParsed ;
  mydata:leftOffice ?leftOfficeParsed ;
  mydata:nominatedBy ?party ;
  mydata:homeState ?stateIRI

```



```

} WHERE {
  # Triple patterns for accessing each row and the columns in contains
  # Note that no triples will be generated for NULL values in the table
  # You should inspect your data in Refine mode and add OPTIONAL accordingly
  ?row a mydata:Row ;
  mydata:Presidency ?Presidency ;
  mydata:President ?President ;
  mydata:Wikipedia_Entry ?Wikipedia_Entry ;
  mydata:Took_office ?Took_office ;
  mydata:Left_office ?Left_office ;
  mydata:Party ?Party ;
  mydata:Portrait ?Portrait ;
  mydata:Thumbnail ?Thumbnail ;
  mydata:Home_State ?Home_State .

  # Uses SPIN function to parse the dates
  bind(spif:parseDate(?Took_office, "dd/MM/yyyy") as ?tookOfficeParsed)
  bind(spif:parseDate(?Left_office, "dd/MM/yyyy") as ?leftOfficeParsed)
  # Uses several functions to construct IRIs for the presidents and their states
  bind(iri(concat("http://example.com/", spif:encodeURL(?President))) as ?presidentIRI)
  bind(iri(concat("http://example.com/", spif:encodeURL(?Home_State))) as ?stateIRI)
} LIMIT 100

```

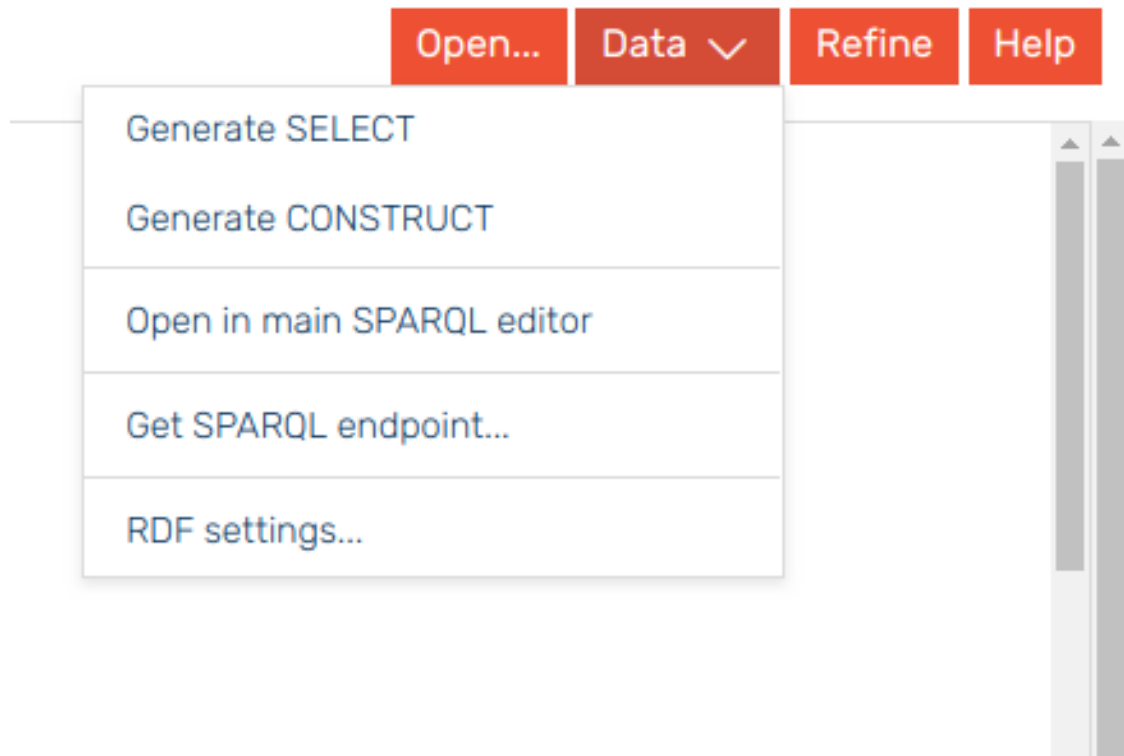
The rows and columns from the table are mapped with the help of the following URIs:

- mydata:Row - the RDF type for each row;
- mydata:rowNumber - the property for row number as an integer literal;
- mydata:<column_name> - properties for each column

5.1.4.6 Importing data in GraphDB

Once you are satisfied with the transformation of your data, you can import it in the current repository without leaving the GraphDB Workbench.

1. Click *Data* -> *Open in main SPARQL editor*



2. The query is the same as the previous one, only with the addition of a SERVICE clause. You only have to change CONSTRUCT to INSERT and remove the LIMIT. Instead of showing the RDF, GraphDB will insert it into the current repository.

SPARQL Query & Update

```

Unnamed X  OntoRefine: USPresident W... X  +
5      INSERT {
6          ?presidentIRI a mydata:President ;
7              mydata:tookOffice ?tookOfficeParsed ;
8              mydata:leftOffice ?leftOfficeParsed ;
9              mydata:nominatedBy ?party ;
10             mydata:homeState ?stateIRI
11     } WHERE {
12     SERVICE <ontorefine:2060364645936> {
13         # Triple patterns for accessing each row and the columns in contains
14         # Note that no triples will be generated for NULL values in the table
15         # You should inspect your data in Refine mode and add OPTIONAL accordingly
16         ?row a mydata:Row ;
17             mydata:Presidency ?Presidency ;
18             mydata:President ?President ;
19             mydata:Wikipedia_Entry ?Wikipedia_Entry ;
20             mydata:Took_office ?Took_office ;
21             mydata:Left_office ?Left_office ;
22             mydata:Party ?Party ;
23             mydata:Portrait ?Portrait ;
24             mydata:Thumbnail ?Thumbnail ;
25             mydata:Home_State ?Home_State .
26
27         # Uses SPIN function to parse the dates
28         bind(spif:parseDate(?Took_office, "dd/MM/yyyy") as ?tookOfficeParsed)
29         bind(spif:parseDate(?Left_office, "dd/MM/yyyy") as ?leftOfficeParsed)
30         # Uses several functions to construct IRIs for the presidents and their states
31         bind(iri(concat("http://example.com/", spif:encodeURL(?President))) as ?presidentIRI)
32         bind(iri(concat("http://example.com/", spif:encodeURL(?Home_State))) as ?stateIRI)
33     }
34 }
35 }
36

```

3. Execute the query to import the results.

5.1.4.7 Rows and records

OpenRefine can view the same data both as rows and as records. See <https://github.com/OpenRefine/OpenRefine/wiki/Variables#record> and <http://kb.refinepro.com/2012/03/difference-between-record-and-row.html>.

Records

For the purpose of the demo, we can create records for each party. Go to *Party* column -> *Edit column* -> *Move column to the beginning*. By clicking *Party* column -> *Text facet*, we can see that some values contain spaces.

Refresh

Reset All

Remove All

Party
change

12 choices
name count
Cluster

Democratic 1

Democratic 4

Democratic 10

Democratic-Republican 3

Democratic-Republican/National Republican 1

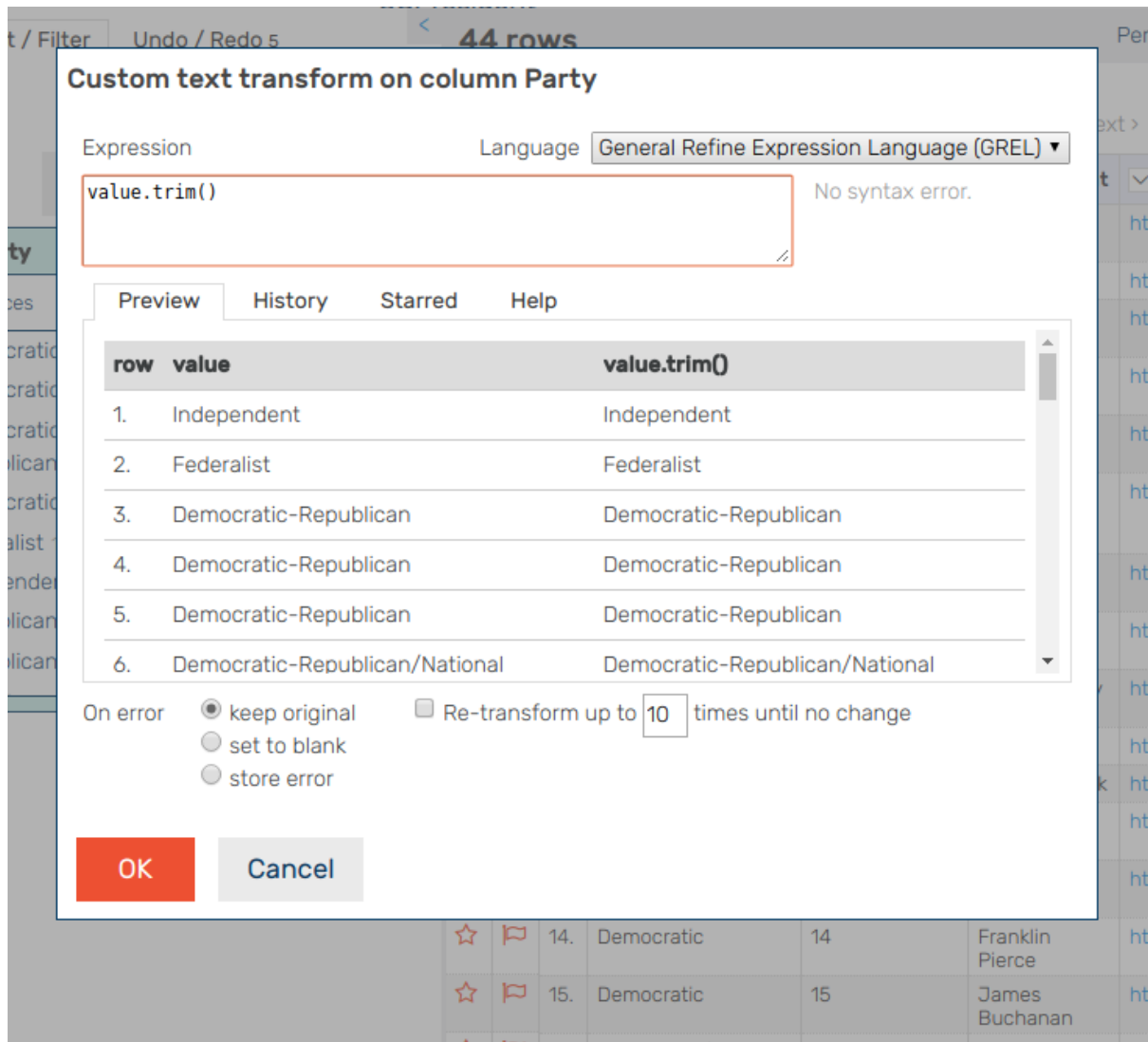
Democratic/National Union 1

Federalist 1

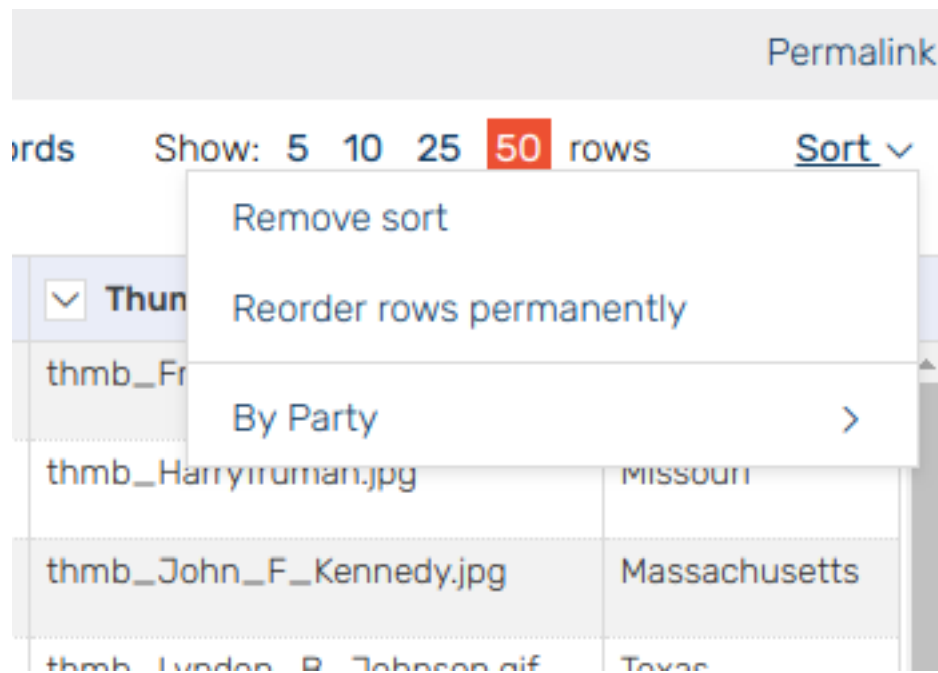
Independent 1

All			Party	
☆	🚩	1.	Independent	
☆	🚩	2.	Federalist	
☆	🚩	3.	Democratic-Republican	
☆	🚩	4.	Democratic-Republican	
☆	🚩	5.	Democratic-Republican	
☆	🚩	6.	Democratic-Republican/National Republican	
☆	🚩	7.	Democratic	
☆	🚩	8.	Democratic	
☆	🚩	9.	Whig	
☆	🚩	10.	Whig	
☆	🚩	11.	Democratic	
☆	🚩	12.	Whig	

1. Clean them by *Party* column -> *Edit cells* -> *Transform* and type value `.trim()`. Observe how the text facet is updated.



2. Now click *Party* column -> *Sort* -> *OK*. Just use the defaults.
3. Then *Sort* -> *Reorder rows permanently*



4. And *Party* column -> *Edit cells* -> *Blank down*. Click on *Records*. The result looks like this:

9 records

Show as: rows **records** Show: 5 10 25 **50** records

☆	▼ All	▼ Party	▼ Presidency	▼ President	▼ Wikipedia Entry	▼ Took office
☆	P	1. Democratic	7	Andrew Jackson	http://en.wikipedia.org/wiki/Andrew_Jackson	4/03/1829
☆	P		8	Martin Van Buren	http://en.wikipedia.org/wiki/Martin_Van_Buren	4/03/1837
☆	P		11	James K. Polk	http://en.wikipedia.org/wiki/James_K._Polk	4/03/1845
☆	P		14	Franklin Pierce	http://en.wikipedia.org/wiki/Franklin_Pierce	4/03/1853
☆	P		15	James Buchanan	http://en.wikipedia.org/wiki/James_Buchanan	4/03/1857
☆	P		22	Grover Cleveland	http://en.wikipedia.org/wiki/Grover_Cleveland	4/03/1885
☆	P		24	Grover Cleveland (2nd term)	http://en.wikipedia.org/wiki/Grover_Cleveland	4/03/1893
☆	P		28	Woodrow Wilson	http://en.wikipedia.org/wiki/Woodrow_Wilson	4/03/1913
☆	P		32	Franklin D. Roosevelt	http://en.wikipedia.org/wiki/Franklin_D._Roosevelt	4/03/1933
☆	P		33	Harry S. Truman	http://en.wikipedia.org/wiki/Harry_S._Truman	12/4/1945
☆	P		35	John F. Kennedy	http://en.wikipedia.org/wiki/John_F._Kennedy	20/01/1961
☆	P		36	Lyndon B. Johnson	http://en.wikipedia.org/wiki/Lyndon_B._Johnson	22/11/1963
☆	P		39	Jimmy Carter	http://en.wikipedia.org/wiki/Jimmy_Carter	20/01/1977
☆	P		42	Bill Clinton	http://en.wikipedia.org/wiki/Bill_Clinton	20/01/1993
☆	P		44	Barack Obama	http://en.wikipedia.org/wiki/Barack_Obama	20/01/2009
☆	P	2. Democratic/National Union	17	Andrew Johnson	http://en.wikipedia.org/wiki/Andrew_Johnson	15/04/1865
☆	P	3. Democratic-Republican	3	Thomas Jefferson	http://en.wikipedia.org/wiki/Thomas_Jefferson	4/03/1801
☆	P		4	James Madison	http://en.wikipedia.org/wiki/James_Madison	4/03/1809
☆	P		5	James Monroe	http://en.wikipedia.org/wiki/James_Monroe	4/03/1817
☆	P	4. Democratic-Republican/National Republican	6	John Quincy Adams	http://en.wikipedia.org/wiki/John_Quincy_Adams	4/03/1825
☆	P	5. Federalist	2	John Adams	http://en.wikipedia.org/wiki/John_Adams	4/03/1797
☆	P	6. Independent	1	George Washington	http://en.wikipedia.org/wiki/George_Washington	30/04/1789
☆	P	7. Republican	18	Ulysses S. Grant	http://en.wikipedia.org/wiki/Ulysses_S._Grant	4/03/1869

5. Now, the *RDF* button leads to the following query and results:

name: USPresident Wikipedia URLs Thumbs HS csv

Open...DataRefineHelp

```
1 PREFIX mydata: <http://example.com/resource/>
2 PREFIX spif: <http://spinrdf.org/spif#>
3
4 # Example query returning RDF data
5 SELECT * {
6   # Triple patterns for accessing each record and the rows it contains
7   ?record a mydata:Record ;
8     mydata:hasRow ?row ;
9     mydata:recordId ?recordId .
10
11   # Triple patterns for accessing each row and the columns it contains
12   # Note that no triples will be generated for NULL values in the table
13   # You should inspect your data in Refine mode and add OPTIONAL accordingly
14   ?row a mydata:Row ;
15     mydata:Presidency ?Presidency ;
16     mydata:President ?President ;
17     mydata:Wikipedia_Entrypoint ?Wikipedia_Entrypoint ;
18     mydata:Took_office ?Took_office ;
19     mydata:Left_office ?Left_office ;
20     mydata:Portrait ?Portrait ;
21     mydata:Thumbnail ?Thumbnail ;
22     mydata:Home_State ?Home_State .
23 }
```

TableRaw ResponsePivot TableGoogle Chart

Filter query resultsShowing 1 to 44 of 44 entries

	record	row	recordId	Presidency	President	Wikipedia_Entrypoint	Took_office	Left_office	Portrait	Thumbnail	Home_State
1	mydata:Record1	mydata:Row1	Democratic	7	Andrew Jackson	http://en.wikipedia.org/wiki/Andrew_Jackson	4/03/1829	4/03/1837	Andrew_Jackson_head.gif	thmb_Andrew_Jackson_head.gif	Tennessee
2	mydata:Record1	mydata:Row2	Democratic	8	Martin Van Buren	http://en.wikipedia.org/wiki/Martin_Van_Buren	4/03/1837	4/03/1841	MartinVanBuren.gif	thmb_MartinVanBuren.gif	New York
3	mydata:Record1	mydata:Row3	Democratic	11	James K. Polk	http://en.wikipedia.org/wiki/James_K_Polk	4/03/1845	4/03/1849	JamesPolk.gif	thmb_JamesPolk.gif	Tennessee
4	mydata:Record1	mydata:Row4	Democratic	14	Franklin Pierce	http://en.wikipedia.org/wiki/Franklin_Pierce	4/03/1853	4/03/1857	FranklinPierce.gif	thmb_FranklinPierce.gif	New Hampshire
5	mydata:Record1	mydata:Row5	Democratic	15	James Buchanan	http://en.wikipedia.org/wiki/James_Buchanan	4/03/1857	4/03/1861	JamesBuchanan.gif	thmb_JamesBuchanan.gif	Pennsylvania

When there are records, the following URIs will be used in addition to the row ones:

- mydata:recordNNN - identifies a record, NNN is the 1-based record number;
- mydata:Record - the RDF type for each record;
- mydata:recordId - the property for record id (the value from the first column that creates an OpenRefine record);
- mydata:hasRow - the property that links records to rows.

Splitting content in a cell

Load the movie dataset for this example. Sometimes you may have multiple values in the same cell. You can easily split them and generate separate URIs using SPIF functions.

```
PREFIX mydata: <http://example.com/resource/>
PREFIX spif: <http://spinrdf.org/spif#>

SELECT * {
  ?row a mydata:Row ;
    mydata:genres ?genres ;
} LIMIT 100
```


OntoRefine name: movie_metadata.csv

Open... Data ▾ Refine Help

```

1 PREFIX mydata: <http://example.com/resource/>
2 PREFIX spif: <http://spinrdf.org/spif#>
3
4 SELECT * {
5   ?row a mydata:Row ;
6       mydata:genres ?genres ;
7 } LIMIT 100

```

Run

Table Raw Response Pivot Table Google Chart

Filter query results

Showing 1 to 100 of 100 entries

	row	genres
1	mydata:Row1	Action Adventure Fantasy Sci-Fi
2	mydata:Row2	Action Adventure Fantasy
3	mydata:Row3	Action Adventure Thriller
4	mydata:Row4	Action Thriller
5	mydata:Row5	Documentary
6	mydata:Row6	Action Adventure Sci-Fi
7	mydata:Row7	Action Adventure Romance
8	mydata:Row8	Adventure Animation Comedy Family Fantasy Musical Romance
9	mydata:Row9	Action Adventure Sci-Fi
10	mydata:Row10	Adventure Family Fantasy Mystery
11	mydata:Row11	Action Adventure Sci-Fi

```

PREFIX mydata: <http://example.com/resource/>
PREFIX spif: <http://spinrdf.org/spif#>

```

```

CONSTRUCT {
  ?m a mydata:Movie ;
    mydata:hasGenre ?genre
} WHERE {
  ?m a mydata:Row ;
    mydata:genres ?genres .
  ?genre spif:split (?genres "\\|")
} LIMIT 100

```

OntoRefine name: movie_metadata.csv

Open... Data ▾ Refine Help

```

1 PREFIX mydata: <http://example.com/resource/>
2 PREFIX spif: <http://spinrdf.org/spif#>
3
4 CONSTRUCT {
5   ?m a mydata:Movie ;
6       mydata:hasGenre ?genre
7 } WHERE {
8   ?m a mydata:Row ;
9       mydata:genres ?genres .
10  ?genre spif:split (?genres "\\|")
11 } LIMIT 100

```

Run

Table Raw Response Pivot Table Google Chart

Filter query results

Showing 1 to 100 of 100 entries

	subject	predicate	object
1	mydata:Row1	mydata:hasGenre	"Action"^^xsd:string
2	mydata:Row1	mydata:hasGenre	"Adventure"^^xsd:string
3	mydata:Row1	mydata:hasGenre	"Fantasy"^^xsd:string
4	mydata:Row1	mydata:hasGenre	"Sci-Fi"^^xsd:string
5	mydata:Row1	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	mydata:Movie
6	mydata:Row10	mydata:hasGenre	"Adventure"^^xsd:string
7	mydata:Row10	mydata:hasGenre	"Family"^^xsd:string
8	mydata:Row10	mydata:hasGenre	"Fantasy"^^xsd:string
9	mydata:Row10	mydata:hasGenre	"Mystery"^^xsd:string
10	mydata:Row10	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	mydata:Movie
11	mydata:Row11	mydata:hasGenre	"Action"^^xsd:string
12	mydata:Row11	mydata:hasGenre	"Adventure"^^xsd:string
13	mydata:Row11	mydata:hasGenre	"Sci-Fi"^^xsd:string

5.1.4.8 Additional resources

- [OpenRefine Documentation](#)
- [OpenRefine Documentation for Users](#)
- [OpenRefine Tutorial](#)
- [Tutorial: OpenRefine, By Atima Han Zhuang Ishita Vedvyas Rishikesh Dole](#)
- [Google Refine Tutorial, by David Huynh, Ph.D.](#)

5.2 Exploring data

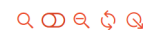
What's in this document?

- *Class hierarchy*
 - *Explore your data - different actions*
 - *Domain-range graph*
- *Class relationships*
- *Explore resources*
 - *Explore resources through the easy graph*
 - *Create your own visual graph*
 - *Saved graphs*
- *Viewing and editing resources*
 - *View and add a resource*
 - *Edit a resource*

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.



- 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.

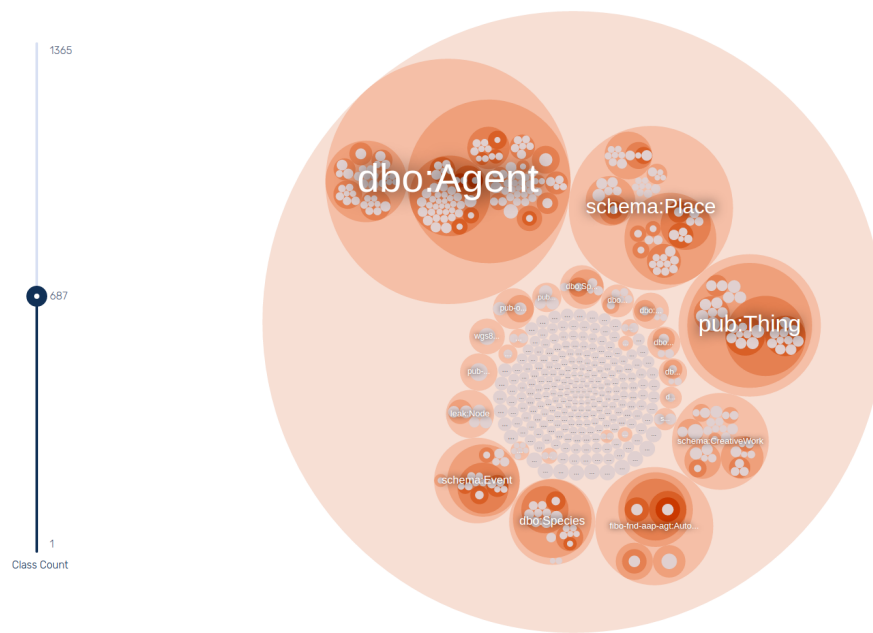
Anacostia High School

Source: http://dbpedia.org/resource/Anacostia_High_School

subject	predicate	object	context	all
Explicit only	Show Blank Nodes	Download as		
subject	predicate	object	context	
1 dbr:Anacostia_High_School	dbo:abstract	Anacostia High School is a public high school located in the Southeast quadrant of the District of Columbia.	onto:explicit	
2 dbr:Anacostia_High_School	dbo:address	1601 16th Street Southeast	onto:explicit	
3 dbr:Anacostia_High_School	dbo:campusType	UrbanWard 8	onto:explicit	
4 dbr:Anacostia_High_School	dbo:city	dbr:Washington,_DC	onto:explicit	
5 dbr:Anacostia_High_School	dbo:district	dbr:District_of_Columbia_Public_Schools	onto:explicit	
6 dbr:Anacostia_High_School	dbo:facultySize	"64"^^xsd:nonNegativeInteger	onto:explicit	
7 dbr:Anacostia_High_School	dbo:foundingYear	"1937"^^xsd:gYear	onto:explicit	
8 dbr:Anacostia_High_School	dbo:grades	dbr:Ninth_grade	onto:explicit	
9 dbr:Anacostia_High_School	dbo:mascot	"Indians"^^xsd:string	onto:explicit	
10 dbr:Anacostia_High_School	dbo:numberOfStudents	"836"^^xsd:nonNegativeInteger	onto:explicit	
11 dbr:Anacostia_High_School	dbo:postalCode	"20020"^^xsd:string	onto:explicit	
12 dbr:Anacostia_High_School	dbo:ratio	"13.06"^^xsd:string	onto:explicit	
13 dbr:Anacostia_High_School	dbo:state	dbr:District_of_Columbia	onto:explicit	
14 dbr:Anacostia_High_School	dbo:thumbnail	http://commons.wikimedia.org/wiki/Special:FilePath/AnacostiaHS_DC.jpg?width=300	http://dbpedia.org/data	
15 dbr:Anacostia_High_School	dbo:type	dbr:Public_high_school	onto:explicit	
16 dbr:Anacostia_High_School	dbo:wikiPageExternalLink	http://profiles.dcps.dc.gov/Anacostia	http://dbpedia.org/data	
17 dbr:Anacostia_High_School	dbo:wikiPageID	"26988334"^^xsd:integer	http://dbpedia.org/data	
18 dbr:Anacostia_High_School	dbo:wikiPageLength	"3622"^^xsd:nonNegativeInteger	http://dbpedia.org/data	
19 dbr:Anacostia_High_School	dbo:wikiPageOutDegree	"24"^^xsd:nonNegativeInteger	http://dbpedia.org/data	
20 dbr:Anacostia_High_School	dbo:wikiPageRevisionID	"669422347"^^xsd:integer	http://dbpedia.org/data	

- 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*.

Class hierarchy ⓘ

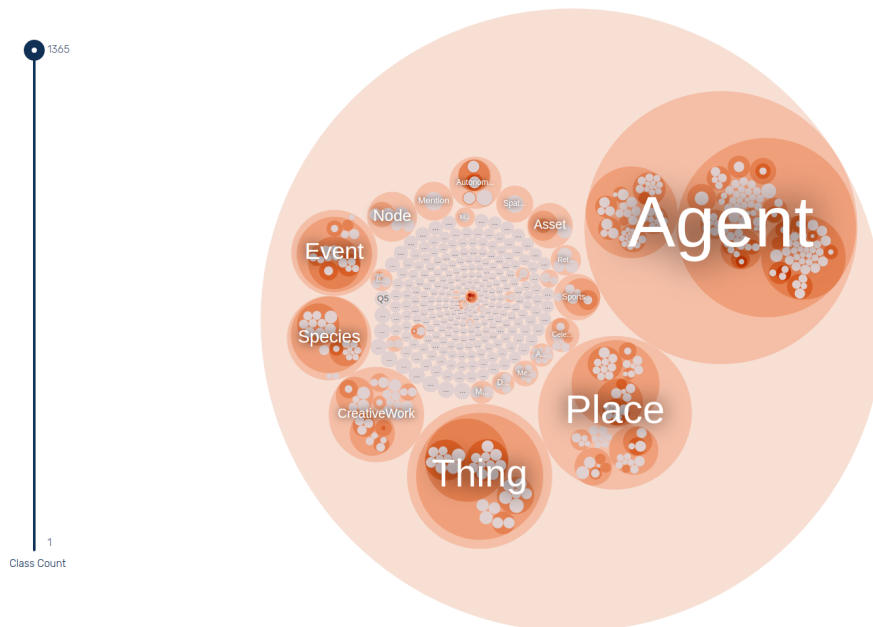


- To administer your data view, use the toolbar options on the right-hand side of the screen.



- 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.

Class hierarchy ⓘ

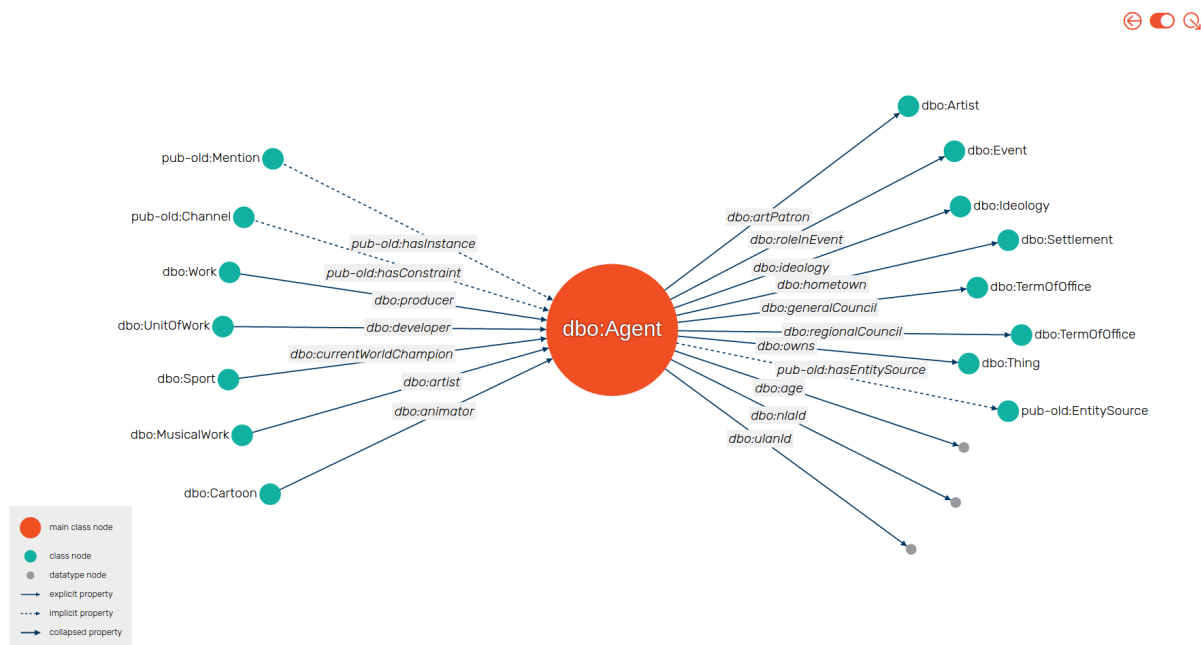


- To zoom out of a particular class, click the *Focus diagram* icon.
- To reload the data on the diagram, click the *Reload diagram* icon. This is recommended when you have updated the data in your repository, or when you are experiencing some strange behavior, for example you cannot see a given class.
- To export the diagram as an .svg image, click the *Export Diagram* download icon.

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*).

Domain-Range graph ⓘ



- To administer your graph view, use the toolbar options on the right-hand side of the screen.

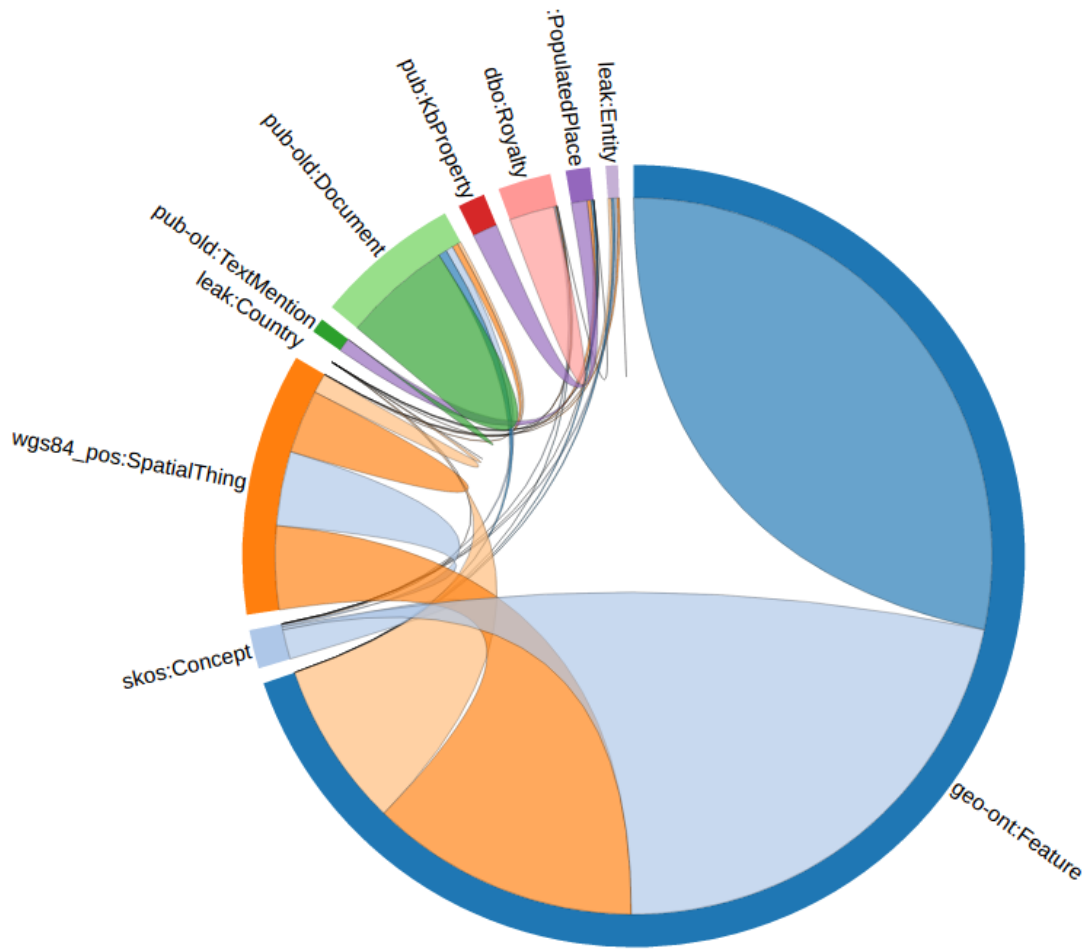


- To go back to your class in the RDF Class hierarchy, click the *Back to Class hierarchy diagram* button.
- To export the diagram as an .svg image, click the *Export Diagram* download icon.

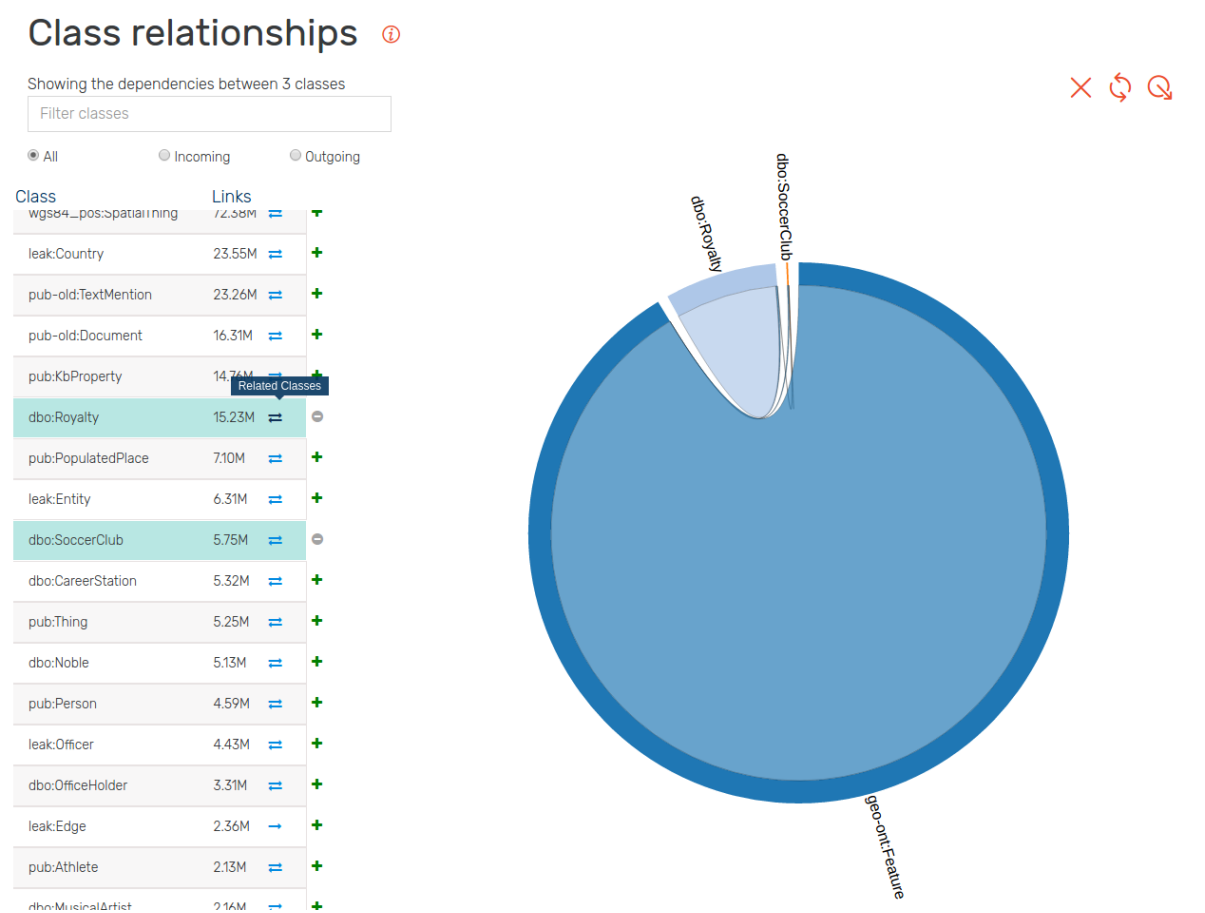
5.2.2 Class relationships

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**.

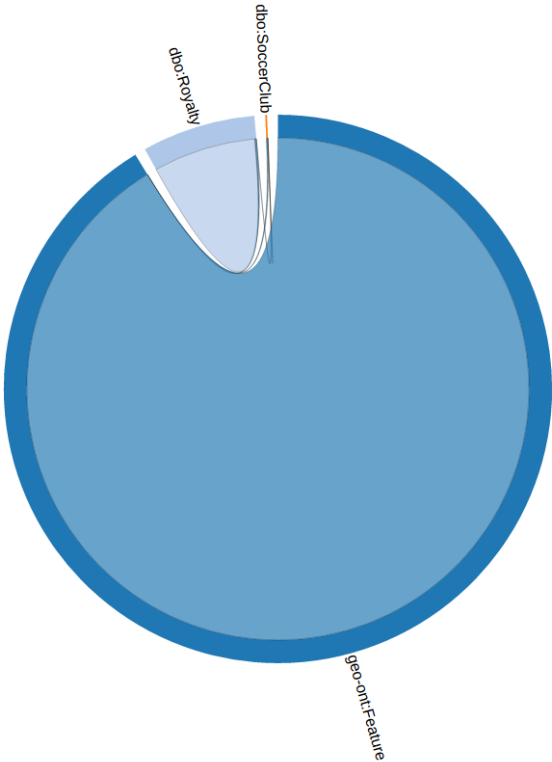
Class relationships ⓘ

Showing the dependencies between 3 classes

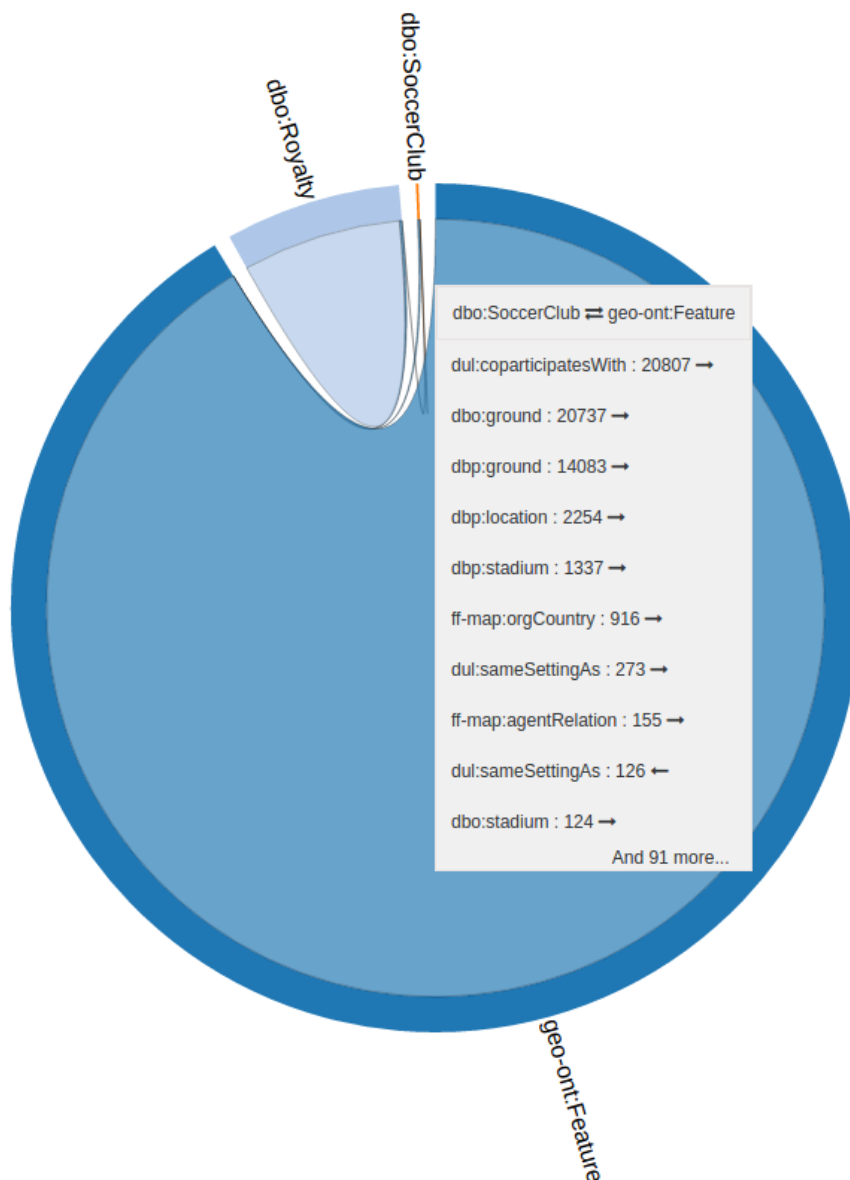
Filter classes

☒ All ☐ Incoming ☐ Outgoing

Class	Links	
wgse4_pos:spatial thing	72.58M	+
leak:Country	23.55M	+
pub-old:TextMention	23.26M	+
pub-old:Document	16.31M	+
pub:KbProperty	14.76M	+
dbo:Royalty	15.23M	+
pub:PopulatedPlace	7.10M	+
leak:Entity	6.31M	+
dbo:SoccerClub	5.75M	+
dbo:CareerStation	5.32M	+
pub:Thing	5.25M	+
dbo:Noble	5.13M	+
pub:Person	4.59M	+
leak:Officer	4.43M	+
dbo:OfficeHolder	3.31M	+
leak:Edge	2.36M	+
pub:Athlete	2.13M	+
dbo:MusicalArtist	2.16M	+



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.



Please note that all 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.

5.2.3 Explore resources

5.2.3.1 Explore resources through the easy graph

Navigate to *Explore* -> *Visual graph*. *Easy graph* gives you the opportunity 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.

Visual graph ⓘ



<http://dbpedia.org/resource/Sofia>

http://dbpedia.org/resource/Sofia_Province

http://data.ontotext.com/publishing/person/Sofia_Richie

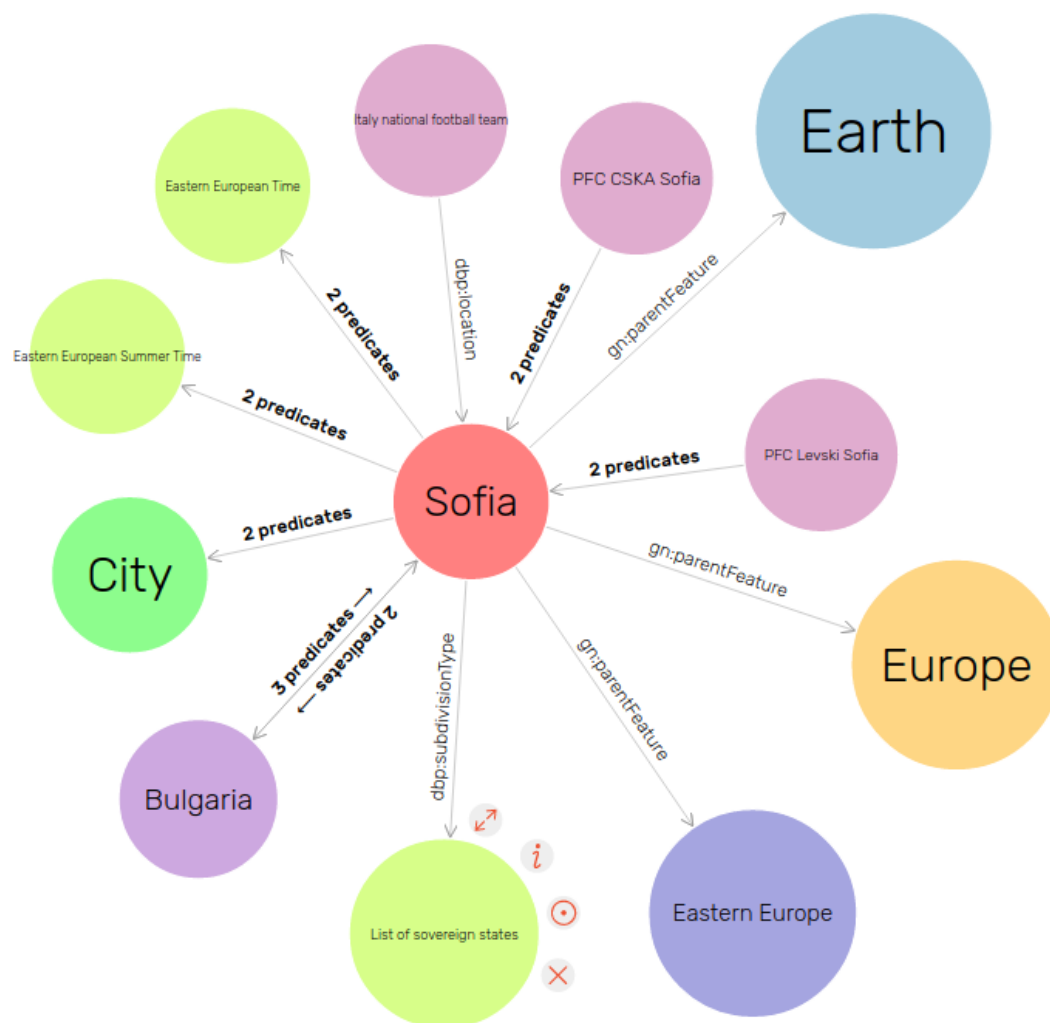
http://dbpedia.org/resource/Sofia_University

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 (*/ˈsoʊfiə/*) (Bulgarian: *София*, *Sofiya*, pronounced *[ˈsɔfijɐ]*) is the capital and largest city of Bulgaria. Sofia is the 14th largest city in the European Union with 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.

[Hide full comment](#)

Sofia • Sofia^{en}

Types:

dbo:City

schema:City

gn:Feature

geo-pos:SpatialThing

wd:Q515

RDF rank:

0.26



Search instance properties

dbo:PopulatedPlace/areaTotal

492.0

dbo:PopulatedPlace/populationDensity

2406.0

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.

Graph Settings



Maximum Links to Show

20

Preferred languages

en Add a language tag

☒ Show Predicate Labels

Types

Predicates

Preferred types

http://dbpedia.org/ontology/Person
Add preferred type

☒ Show Preferred Types Only

Ignored types

Add ignored type

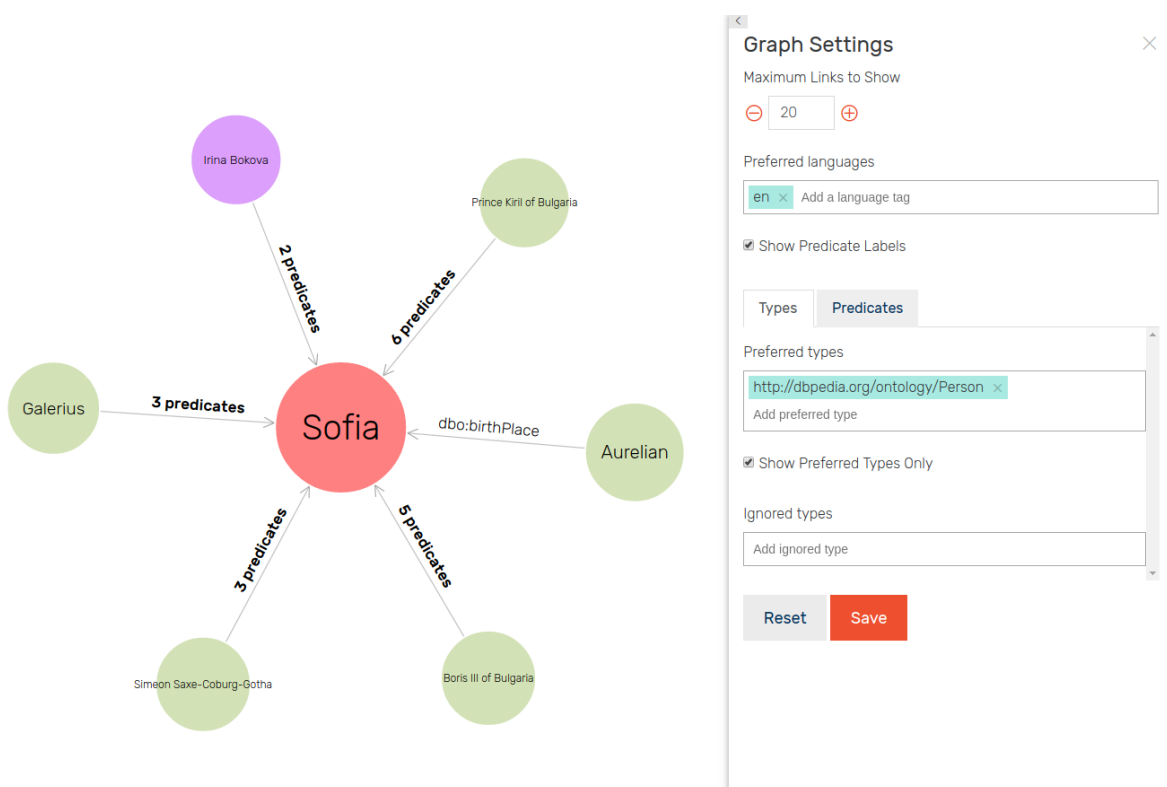
Reset

Save

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> 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)

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.

Save

Next 

Cancel

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 Saved graphs

During graph exploration, you can save a snapshot of the graph state 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.

5.2.4 Viewing and editing resources

5.2.4.1 View and add a resource

To view a resource in the repository, go to the GraphDB home page, and start typing in the *Explore -> View resource* field. The retrieved results can be in text format or in the form of a visual graph, with the latter automatically taking you to the *Visual graph* view. You can also navigate to a resource by clicking the SPARQL results links in the table that is returned below.

GraphDB Enterprise

View resource

wine

Text Visual

http://example.com/wine

wine <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#wine>

http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine

Wine Ontology <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine>

http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#Winery

http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WineBody

Active repository

Saved SPARQL queries

Viewing resources provides an easy way to see triples where a given IRI is the subject, predicate or object.

CabernetFrancGrape

Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#CabernetFrancGrape>

subject	predicate	object	context	all
Explicit only	Show Blank Nodes	Download as	Visual graph	
subject	predicate	object	context	
1 vin:CabernetFrancGrape	rdf:type	owl:Thing	http://example.com/wine	
2 vin:CabernetFrancGrape	rdf:type	owl:Thing	http://www.ontotext.com/explicit	
3 vin:CabernetFrancGrape	rdf:type	vin:WineGrape	http://example.com/wine	
4 vin:CabernetFrancGrape	rdf:type	vin:WineGrape	http://www.ontotext.com/explicit	

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.

View resource

Text

Visual

MavrudGrape Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#MavrudGrape>

subject

predicate

object

context

all

Explicit only

Show Blank Nodes

Download as

Visual graph

subject	predicate	object	context
No data available in table			

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 Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#MavrudGrape>

Predicate	Object	Context
Example: 'http://exampleuri.com/examplepath' wgs rdf owl gn xsd fn vin rdfs sesame food	URI value wgs rdf owl gn xsd fn vin rdfs sesame food	Example: 'http://exampleuri.com/examp wgs rdf owl gn xsd fn vin rdfs sesame food 
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WineGrape	 
		View TriG Save

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

Edit MavrudGrape Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#MavrudGrape>

Predicate	Object	Context
Example: 'http://exampleuri.com/examplepath' wgs rdf owl gn xsd fn vin rdfs sesame food	URI value wgs rdf owl gn xsd fn vin rdfs sesame food	Example: 'http://exampleuri.com/examp wgs rdf owl gn xsd fn vin rdfs sesame food 
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#WineGrape	 
		View TriG Save

View the resource graph in TriG format

View TriG

Save

View Statements in TriG

```
{
  <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#MavrudGrape> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
}
```

Cancel

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

Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#SweetRiesling>

Predicate	Object	Context
Example: "http://exampleuri.com/examplepath" wgs rdf owl gn xsd fn vin rdfs sesame food	URI value wgs rdf owl gn xsd fn vin rdfs sesame food	Example: "http://exampleuri.com/exam wgs rdf owl gn xsd fn vin rdfs sesame food
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#Class	http://example.com/wine
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#Class	
http://www.w3.org/2000/01/rdf-schema#subClassOf	...node537	http://example.com/wine
http://www.w3.org/2000/01/rdf-schema#subClassOf	...node538	http://example.com/wine
http://www.w3.org/2000/01/rdf-schema#subClassOf	...node79	
http://www.w3.org/2000/01/rdf-schema#subClassOf	...node80	
http://www.w3.org/2000/01/rdf-schema#subClassOf	http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#DessertWine	http://example.com/wine
http://www.w3.org/2000/01/rdf-schema#subClassOf	http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#DessertWine	

Note: You cannot change or delete the inferred statements.

5.3 Querying Data

What's in this document?

- *Save and share queries*
- *Interrupt queries*

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

Hint: SPARQL is an SQL-like query language for RDF graph databases with the following types:

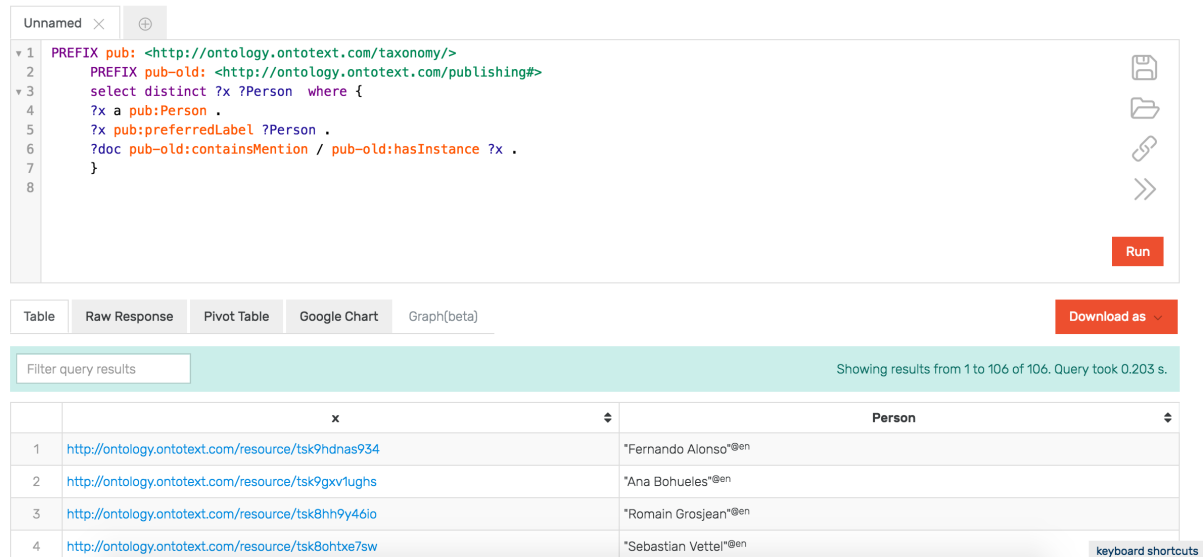
- SELECT - returns tabular results;
- CONSTRUCT - creates a new RDF graph based on query results;
- ASK - returns “YES”, if the query has a solution, otherwise “NO”;
- DESCRIBE - returns RDF data about a resource; useful when you do not know the RDF data structure in the data source;
- INSERT - inserts triples into a graph;

- DELETE - deletes triples from a graph.

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

SPARQL Query & Update

Editor only Editor and results Results only



Unnamed x

```

1 PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
2 PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
3 select distinct ?x ?Person where {
4   ?x a pub:Person .
5   ?x pub:preferredLabel ?Person .
6   ?doc pub-old:containsMention / pub-old:hasInstance ?x .
7 }
8

```

Run

Table Raw Response Pivot Table Google Chart Graph(beta) Download as

Filter query results Showing results from 1 to 106 of 106. Query took 0.203 s.

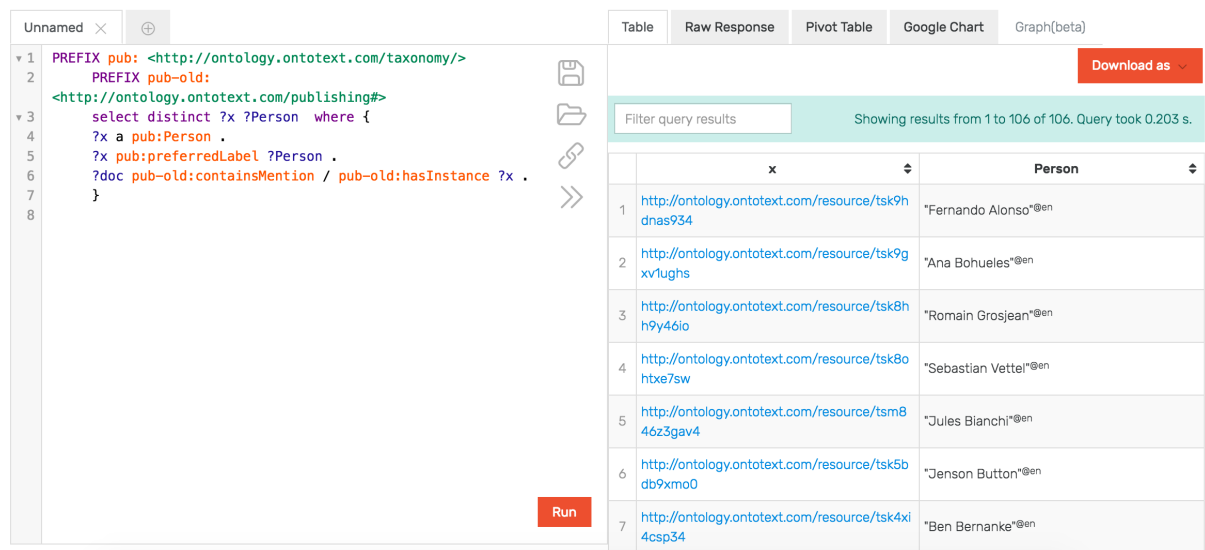
	x	Person
1	http://ontology.ontotext.com/resource/tsk9hdnas934	"Fernando Alonso"@en
2	http://ontology.ontotext.com/resource/tsk9gxv1ugh	"Ana Bohueles"@en
3	http://ontology.ontotext.com/resource/tsk8hh9y46io	"Romain Grosjean"@en
4	http://ontology.ontotext.com/resource/tsk8ohxe7sw	"Sebastian Vettel"@en

keyboard shortcuts

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.

SPARQL Query & Update

Editor only Editor and results Results only



Unnamed x

```

1 PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
2 PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
3 select distinct ?x ?Person where {
4   ?x a pub:Person .
5   ?x pub:preferredLabel ?Person .
6   ?doc pub-old:containsMention / pub-old:hasInstance ?x .
7 }
8

```

Run

Table Raw Response Pivot Table Google Chart Graph(beta) Download as

Filter query results Showing results from 1 to 106 of 106. Query took 0.203 s.

	x	Person
1	http://ontology.ontotext.com/resource/tsk9hdnas934	"Fernando Alonso"@en
2	http://ontology.ontotext.com/resource/tsk9gxv1ugh	"Ana Bohueles"@en
3	http://ontology.ontotext.com/resource/tsk8hh9y46io	"Romain Grosjean"@en
4	http://ontology.ontotext.com/resource/tsk8ohxe7sw	"Sebastian Vettel"@en
5	http://ontology.ontotext.com/resource/tsm846z3gav4	"Jules Bianchi"@en
6	http://ontology.ontotext.com/resource/tsk5bdb9xmo0	"Jenson Button"@en
7	http://ontology.ontotext.com/resource/tsk4xl4csp34	"Ben Bernanke"@en

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 auto-completion 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 the same color (*ON*), otherwise the left element is dark and the right one is greyed out (*OFF*).

3. Execute the query by clicking the *Run* button or use **Ctrl/Cmd + Enter**.

Tip: You can find other useful shortcuts in the *keyboard shortcuts* link in the lower right corner of the SPARQL editor.

4. 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`.

5. 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.
6. 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.

Create New Saved Query

☒ Share query with other users

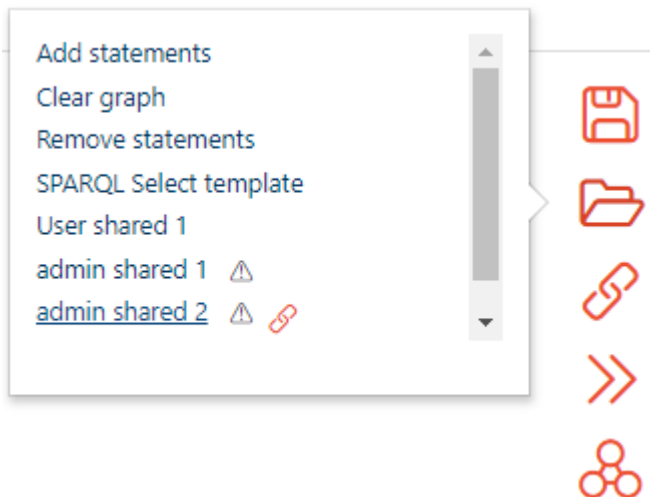
```
select * where {  
  ?s ?p ?o .  
}  
limit 100
```

Cancel

Create

When security is ON in *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.

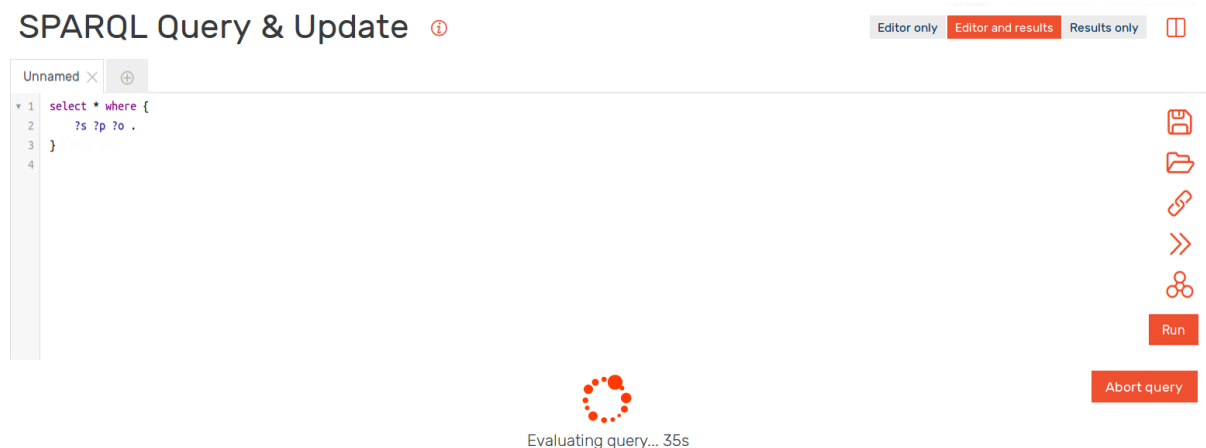


Copy your query as a URL by clicking the *Get URL* to current query icon.

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.



5.4 Exporting Data

What's in this document?

- *Exporting a repository*
- *Exporting individual graphs*
- *Exporting query results*
- *Exporting resources*

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* button and then the format that fits your needs.

Graphs overview ⓘ

Showing 1 - 5 of 5 results Graphs per page: **All** ▾

🔍 Export repository ▾
🗑️ Clear repository

Graphs
The default graph 🔍 ▾ 🗑️
http://example.com/wine 🔍 ▾ 🗑️
http://example.com/movies 🔍 ▾ 🗑️
http://example.com/news 🔍 ▾ 🗑️
http://example.com/publications 🔍 ▾ 🗑️

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.

Graphs overview ⓘ

Showing 1 - 5 of 5 results Graphs per page: **All** ▾

🔍 Export repository ▾
🗑️ Clear repository

Graphs
The default graph 🔍 ▾ 🗑️
http://example.com/wine 🔍 ^ 🗑️
http://example.com/movies 🔍 ▾ 🗑️
http://example.com/news 🔍 ▾ 🗑️
http://example.com/publications 🔍 ▾ 🗑️

JSON
JSON-LD
RDF/XML
N3
N-Triples
N-Quads
Turtle
TriX
TriG
Binary RDF

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:

SweetRiesling

Source: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#SweetRiesling>

subject	predicate	object	context	all
Explicit only				
Show Blank Nodes				
Download as				
JSON				
JSON-LD				
RDF/XML				
N3				
N-Triples				
N-Quads				
Turtle				
TriX				
TriG				
Binary RDF				
Visual graph				
subject	predicate	object	context	
1 vin:SweetRiesling	rdf:type	owl:Class	http://example.com/movies	
2 vin:SweetRiesling	rdf:type	owl:Class	http://example.com/news	
3 vin:SweetRiesling	rdf:type	owl:Class	http://example.com/publications	
4 vin:SweetRiesling	rdf:type	owl:Class	http://example.com/wine	
5 vin:SweetRiesling	rdf:type	owl:Class	http://www.ontotext.com/explicit	
6 vin:SweetRiesling	rdfs:subClassOf	_:node1453	http://example.com/news	
7 vin:SweetRiesling	rdfs:subClassOf	_:node1454	http://example.com/news	
8 vin:SweetRiesling	rdfs:subClassOf	_:node1911	http://example.com/publications	
9 vin:SweetRiesling	rdfs:subClassOf	_:node1912	http://example.com/publications	
10 vin:SweetRiesling	rdfs:subClassOf	_:node537	http://example.com/wine	
11 vin:SweetRiesling	rdfs:subClassOf	_:node538	http://example.com/wine	
12 vin:SweetRiesling	rdfs:subClassOf	_:node79	http://www.ontotext.com/explicit	
13 vin:SweetRiesling	rdfs:subClassOf	_:node80	http://www.ontotext.com/explicit	
14 vin:SweetRiesling	rdfs:subClassOf	_:node995	http://example.com/movies	
15 vin:SweetRiesling	rdfs:subClassOf	_:node996	http://example.com/movies	

5.5 Reasoning

What's in this document?

- *Logical formalism*
- *Rule format and semantics*
- *The ruleset file*
 - *Prefices*
 - *Axioms*
 - *Rules*
- *Rulesets*
 - *Predefined rulesets*
 - *Custom rulesets*
- *Inference*
 - *Reasoner*

- *Rulesets execution*
- *Retraction of assertions*
- *How To's*
 - *Operations on rulesets*
 - *Reinferring*

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.

5.5.1 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*.

5.5.2 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 modelled as rules with empty bodies.

5.5.3 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 Prefices 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.

5.5.3.1 Prefices

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 prefices 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#>
}
```

5.5.3.2 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>
}
```

Note: Axiomatic statements are considered to be inferred for the purpose of query answering because they are a result of semantic interpretation defined by the chosen ruleset.

5.5.3.3 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, predicate, and object can each be a variable, blank node, literal, full URI or the 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) 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

x p y
p <rdf:type> <owl:FunctionalProperty>
x p z
-----
y <owl:sameAs> z

x p z
p <rdf:type> <owl:FunctionalProperty>
x p y
-----
y <owl:sameAs> z

```

Here, the last two variants are identical apart from the rotation of variables y and z, so one of these variants is not needed. The use of the Cut operator above tells the rule compiler to eliminate this last variant, 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
-----
start pc last          [Context <onto:_checkChain>]
```

The RIF rules that implement prp-spo2 use a relation (unrelated to the input or generated triples) called `_checkChain`. The GraphDB implementation maps this relation to the ‘invisible’ context of the same name with the addition of `[Context <onto:_checkChain>]` to certain statement patterns. Generated statements with this context can only be used for bindings to rule premises when the exact same context is specified in the rule premise. The generated statements with this context will not be used for any other rules.

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 addition 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 OWL2RL

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.
-

5.5.4 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.

5.5.4.1 Predefined rulesets

The pre-defined rulesets provided with GraphDB cover various well-known knowledge representation formalisms, and are layered in such a way that each extends the preceding one.

Rule-set	Description
empty	No reasoning, i.e., GraphDB operates as a plain RDF store.
rdfs	Supports the standard model-theoretic RDFS semantics. This includes support for <code>subClassOf</code> and related type inference, as well as <code>subPropertyOf</code> .
rdfs plus	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> .
owl-horst	OWL dialect close to OWL Horst - essentially pD*
owl-max	RDFS and that part of OWL Lite that can be captured in rules (deriving functional and inverse functional properties, all-different, subclass by union/enumeration; min/max cardinality constraints, etc.).
owl2-ql	The OWL2 QL profile - a fragment of OWL2 Full designed so that sound and complete query answering is LOGSPACE with respect to the size of the data. This OWL2 profile is based on DL-LiteR, a variant of DL-Lite that does not require the unique name assumption.
owl2-rl	The OWL2 RL profile - an expressive fragment of OWL2 Full that is amenable for implementation on rule engines.

Note: Not all rulesets support data-type reasoning, which is the main reason why OWL-Horst is not the same as pD*. The ruleset you need to use for a specific repository is defined through the *ruleset parameter*. There are optimized versions of all rulesets that avoid some little used inferences.

Note: The default ruleset is RDFS-Plus (optimized).

OWL2 QL non-conformance

The implementation of OWL2 QL is non-conformant with the W3C OWL2 profiles recommendation as shown in the following table:

Conformant behavior	Implemented behavior
Given a list of disjoint (data or object) properties and an entity that is related with these properties to objects {a, b, c, d, ...}, infer an <code>owl:AllDifferent</code> restriction on an anonymous list of these objects.	For each pair {p, q} ($p \neq q$) of disjoint (data or object) properties, infer the triple: p <code>owl:propertyDisjointWith</code> q Which is more likely to be useful for query answering.
For each class C in the knowledge base, infer the existence of an anonymous class that is the union of a list of classes containing only C.	Not supported. Even if this infinite expansion were possible in a forward chaining rule-based implementation, the resulting statements are of no use during query evaluation.
If a instance of C1, and b instance of C2, and C1 and C2 disjoint, infer: a <code>owl:differentFrom</code> b	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 C1 and C2, and C1 and C2 disjoint, then inconsistent.

5.5.4.2 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.

5.5.5 Inference

5.5.5.1 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.

5.5.5.2 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*).

5.5.5.3 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:

```
[ ] <http://www.ontotext.com/owlim/system#schemaTransaction> [ ]
```

GraphDB will use the smooth-delete algorithm, but will also traverse read-only statements and allow them to be deleted/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:

```
DELETE {
  [[schema statements to delete]]
}
INSERT {
  [ ] <http://www.ontotext.com/owlim/system#schemaTransaction> [ ] .
  [[schema statements to insert]]
}
WHERE { }
```

5.5.6 How To's

5.5.6.1 Operations on rulesets

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

```
prefix sys: <http://www.ontotext.com/owlim/system#>
```

Add a custom ruleset from .pie file

The predicate `sys:addRuleset` adds a custom ruleset from the specified .pie file. The ruleset is named after the filename, without the .pie extension.

Example 1 This creates a new ruleset 'test'. If the absolute path to the file resides on, for example, `/opt/rules/test.pie`, it can be specified as `<file:/opt/rules/test.pie>`, `<file://opt/rules/test.pie>`, or `<file:///opt/rules/test.pie>`, i. e., with 1, 2, or 3 slashes. Relative paths are specified without the slashes or with a dot between the slashes: `<file:opt/rules/test.pie>`, `<file:./opt/rules/test.pie>`, `<file:///./opt/rules/test.pie>`, or even `<file:./opt/rules/test.pie>` (with a dot in front of

the path). Relative paths can be used if you know the work directory of the Java process in which GraphDB runs.

```
INSERT DATA {
  _:b sys:addRuleset <file:c:/graphdb/test-data/test.pie>
}
```

Example 2 Same as above but creates a ruleset called ‘custom’ out of the test.pie file found in the given absolute path.

```
INSERT DATA {
  <_:custom> sys:addRuleset <file:c:/graphdb/test-data/test.pie>
}
```

Example 3 Retrieves the .pie file from the given URL. Again, you can use <_:custom> to change the name of the ruleset to “custom” or as necessary.

```
INSERT DATA {
  _:b sys:addRuleset <http://example.com/test-data/test.pie>
}
```

Add a built-in ruleset

The predicate sys:addRuleset adds 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
    '''Prefixes { a : http://a/ }
    Axioms {}
    Rules
    {
      Id: custom
      a b c
      a <a:custom1> c
      -----
      b <a:custom1> a
    }'''
}
```

Note: Effects on the axiom set

When dealing with more than one ruleset, the result set of axioms is the UNION of all axioms of rulesets added so far. 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 graphdb-ruleset-usage-optimisation)

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”.

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

Delete a ruleset

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

Example

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

Note: Effects on the axiom set when removing a ruleset

When removing a ruleset, 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

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

5.5.6.2 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 reinheritence with an Update statement such as:

```
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*.

5.6 Using the Workbench REST API

What's in this document?

- *Security management*
- *Location management*
- *Repository management*
- *Cluster management*
- *Data import*
- *Saved queries*

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 six major categories:

5.6.1 Security management

Use the security management API to add, edit, or remove users, thus integrating the Workbench security into an existing system.

5.6.2 Location management

Use the location management API to attach, activate, edit, or detach locations.

5.6.3 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.

5.6.4 Cluster management

Use the cluster management API to connect or disconnect workers to/from masters, masters to/from masters and to query the status of connected workers. You can also trigger a backup or restore on any master node. The advantage of using the cluster management API is not having to work with JMX. When combined with location and repository management, it can be used to automate the setup of a GraphDB EE cluster.

5.6.5 Data import

Use the data import API to import data in GraphDB. You can choose between server files and a remote URL.

5.6.6 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.

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

import-controller : Data import	Show/Hide	List Operations	Expand Operations
saved-queries-controller : Saved queries	Show/Hide	List Operations	Expand Operations
security-management-controller : Security management	Show/Hide	List Operations	Expand Operations
cluster-management-controller : Cluster management	Show/Hide	List Operations	Expand Operations
location-management-controller : Location management	Show/Hide	List Operations	Expand Operations
repository-management-controller : Repository management	Show/Hide	List Operations	Expand Operations

[BASE URL: /]

RDF4J API

repositories : Repository management	Show/Hide	List Operations	Expand Operations
sparql : SPARQL	Show/Hide	List Operations	Expand Operations
contexts : Contexts management	Show/Hide	List Operations	Expand Operations
namespaces : Namespaces management	Show/Hide	List Operations	Expand Operations
graph-store : Graph Store protocol	Show/Hide	List Operations	Expand Operations
transactions : Transactions management	Show/Hide	List Operations	Expand Operations
protocol : Protocol verification	Show/Hide	List Operations	Expand Operations

[BASE URL: / , API VERSION: 2.7.8]

Click on a functionality group to expand it and see the operations that it includes. Click on an operation to see details about it.

GraphDB Workbench API

import-controller : Data import [Show/Hide](#) [List Operations](#) [Expand Operations](#)

DELETE </rest/data/import/server/{repository}> [Cancel server file import operation](#)

Response Class (Status 200)
string

Response Content Type [*/*](#)

Parameters

Parameter	Value	Description	Parameter Type	Data Type
repository	(required)	repository	path	string
name		The filename import to interrupt.	query	string

[Try it out!](#)

GET </rest/data/import/server/{repository}> [Get server files available for import](#)

POST </rest/data/import/server/{repository}> [Import a server file into the repository](#)

POST </rest/data/import/upload/{repository}/url> [Import from data URL into the repository](#)

saved-queries-controller : Saved queries [Show/Hide](#) [List Operations](#) [Expand Operations](#)

security-management-controller : Security management [Show/Hide](#) [List Operations](#) [Expand Operations](#)

cluster-management-controller : Cluster management [Show/Hide](#) [List Operations](#) [Expand Operations](#)

location-management-controller : Location management [Show/Hide](#) [List Operations](#) [Expand Operations](#)

repository-management-controller : Repository management [Show/Hide](#) [List Operations](#) [Expand Operations](#)

similarity-controller : Similarity queries [Show/Hide](#) [List Operations](#) [Expand Operations](#)

5.7 Using GraphDB with the RDF4J API

What's in this document?

- *RDF4J API*
 - *Accessing a local repository*
 - *Accessing a remote repository*
- *SPARQL endpoint*
- *Graph Store HTTP Protocol*

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.

5.7.1 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.

5.7.1.1 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`.

```
@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 owl: <http://www.ontotext.com/trree/owlim#>.

[] a rep:Repository ;
  rep:repositoryID "graphdb-repo" ;
  rdfs:label "GraphDB Getting Started" ;
  rep:repositoryImpl [
    rep:repositoryType "openrdf:SailRepository" ;
    sr:sailImpl [
      sail:sailType "graphdb:FreeSail" ;
      owl:ruleset "owl-horst-optimized" ;
      owl:storage-folder "storage" ;
      owl:base-URL "http://example.org/owlim#" ;
      owl:repository-type "file-repository" ;
      owl:imports "./ontology/owl.rdfs" ;
      owl:defaultNS "http://example.org/owlim#" .
    ]
  ] .
```

The Java code that uses the configuration to instantiate a repository and get a connection to it is as follows:

```
// Instantiate a local repository manager and initialize it
RepositoryManager repositoryManager = new LocalRepositoryManager(new File("."));
repositoryManager.initialize();

// Instantiate a repository graph model
TreeModel graph = new TreeModel();

// Read repository configuration file
InputStream config = EmbeddedGraphDB.class.getResourceAsStream("/repo-defaults.ttl");
RDFParser rdfParser = Rio.createParser(RDFFormat.TURTLE);
rdfParser.setRDFHandler(new StatementCollector(graph));
rdfParser.parse(config, RepositoryConfigSchema.NAMESPACE);
config.close();

// Retrieve the repository node as a resource
Resource repositoryNode = GraphUtil.getUniqueSubject(graph, RDF.TYPE, RepositoryConfigSchema.
    REPOSITORY);

// Create a repository configuration object and add it to the repositoryManager
RepositoryConfig repositoryConfig = RepositoryConfig.create(graph, repositoryNode);
repositoryManager.addRepositoryConfig(repositoryConfig);
```

```
// Get the repository from repository manager, note the repository id set in configuration .ttl file
Repository repository = repositoryManager.getRepository("graphdb-repo");

// Open a connection to this repository
RepositoryConnection repositoryConnection = repository.getConnection();

// ... use the repository

// Shutdown connection, repository and manager
repositoryConnection.close();
repository.shutdown();
repositoryManager.shutdown();
```

The 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`.

5.7.1.2 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:

```
RepositoryManager repositoryManager =
    new RemoteRepositoryManager( "http://192.168.1.25:7200" );
repositoryManager.initialize();
```

The rest of the example program should work as expected, although the following library files must be added to the class-path:

- commons-httpclient-3.1.jar
- commons-codec-1.10.jar

5.7.2 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.

5.7.3 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.

5.8 Plugins

5.8.1 Plugin API

What's in this document?

- *What is the GraphDB Plugin API*
- *Description of a GraphDB plugin*
- *The life cycle of a plugin*
 - *Discovery*
 - *Configuration*
 - *Initialization*
 - *Request processing*
 - *Shutdown*
- *Repository internals*
 - *Statements and Entities*
 - *System properties*
 - *Repository properties*
- *Query processing*
 - *Pre-processing*
 - *Pattern interpretation*
 - *Post-processing*
- *Update processing*
 - *Updates involving specific predicates*
 - *Removal of entire contexts*
 - *Intercepting data for specific contexts*
- *Transactions*
- *Exceptions*
- *Accessing other plugins*
- *List of plugin interfaces and classes*
 - *Basics*
 - *Data structures*
 - *Query request handlers*

- *Update request handlers*
- *Notification listeners*
- *Plugin dependencies*
- *Health checks*
- *Exceptions*
- *Putting it all together: example plugins*
 - *ExampleBasicPlugin*
 - *ExamplePlugin*

5.8.1.1 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.

5.8.1.2 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:

```
/**
 * The base interface for a GraphDB plugin. As a minimum a plugin must implement this interface.
 * <p>
 * Plugins also need to be listed in META-INF/services/com.ontotext.trree.sdk.Plugin so that Java's
 * ↪services
 * ↪mechanism may discover them automatically.
 */
public interface Plugin extends Service {
    /**
     * A method used by the plugin framework to configure each plugin's file system directory. This
     * directory should be used by the plugin to store its files
     *
     * @param dataDir file system directory to be used for plugin related files
     */
    void setDataDir(File dataDir);

    /**
     * A method used by the plugin framework to provide plugins with a {@link Logger} object
     *
     * @param logger {@link Logger} object to be used for logging
     */
    void setLogger(Logger logger);

    /**
     * Plugin initialization method called once when the repository is being initialized, after the
     * ↪plugin has been
     * ↪configured and before it is actually used. It enables plugins to execute whatever
     * ↪initialization routines they consider appropriate, load resources, open connections, etc.,
     * ↪based on the
     * ↪specific reason for initialization, e.g. backup.
     * <p>

```

```

    * The provided {@link PluginConnection} instance may be used to create entities needed by the
    ↪ plugin.
    *
    * @param reason          the reason for initialization
    * @param pluginConnection an instance of {@link PluginConnection}
    */
    void initialize(InitReason reason, PluginConnection pluginConnection);

    /**
    * Sets a new plugin fingerprint.
    * Every plugin should maintain a fingerprint of its data that could be used by GraphDB to
    ↪ determine if the
    * data has changed or not. Initially, on system initialization the plugins are injected their
    * fingerprints as they reported them before the last system shutdown
    *
    * @param fingerprint the last known plugin fingerprint
    */
    void setFingerprint(long fingerprint);

    /**
    * Returns the fingerprint of the plugin.
    * <p>
    * Every plugin should maintain a fingerprint of its data that could be used by GraphDB to
    ↪ determine if the
    * data has changed or not. The plugin fingerprint will become part of the repository fingerprint.
    *
    * @return the current plugin fingerprint based on its data
    */
    long getFingerprint();

    /**
    * Plugin shutdown method that is called when the repository is being shutdown. It enables
    ↪ plugins to execute whatever
    * finalization routines they consider appropriate, free resources, buffered streams, etc.,
    ↪ based on the
    * specific reason for shutdown.
    *
    * @param reason the reason for shutdown
    */
    void shutdown(ShutdownReason reason);
}

```

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.tree.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.

```

/**
 * Interface implemented by all run-time discoverable services (e.g. {@link Plugin} instances).
    ↪ Classes
 * implementing this interface should furthermore be declared in the respective
 * META-INF/services/<class.signature> file and will then be discoverable at run-time.
 * <p>
 * Plugins need not implement this interface directly but rather implement {@link Plugin}.
 */
public interface Service {

```

```
/**
 * Gets the service name (serves as a key for discovering the service)
 *
 * @return service name
 */
String getName();
}
```

There are many more functions (interfaces) that a plugin could implement, but these are all optional and are declared in separate interfaces. Implementing any such complementary interface is the means to announce to the system what this particular plugin can do in addition to its mandatory plugin responsibilities. It is then automatically used as appropriate. See *List of plugin interfaces and classes*.

5.8.1.3 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.trree.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 IRIs that will be used to communicate with the plugin.

This phase is also called when a plugin is enabled after repository initialization.

Request processing

The plugin participates in the request processing. The request phase applies to the evaluation of SPARQL queries, `getStatements` calls, the transaction stages and the execution of SPARQL updates. Various event notifications can also be part of this phase.

This phase is optional for the plugins but no plugin is useful without implementing at least one of *its interfaces*.

Request processing can be divided roughly into query processing and update processing.

Query processing

Query processing includes several sub-phases that can be used on their own or combined together:

Pre-processing Plugins are given the chance to modify the request before it is processed. In this phase, they could also initialize a context object, which will be visible 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.

5.8.1.4 Repository internals

The repository internals are accessed via an instance of `PluginConnection`:

```
/**
 * The {@link PluginConnection} interface provides access to various objects that can be used to
 * ↪ query data
 * or get the properties of the current transaction. An instance of {@link PluginConnection} will be
 * ↪ passed to almost
 * all methods that a plugin may implement.
 */
public interface PluginConnection {
    /**
     * Returns an instance of {@link Entities} that can be used to retrieve or create RDF entities.
     *
     * @return an {@link Entities} instance
     */
    Entities getEntities();

    /**
     * Returns an instance of {@link Statements} that can be used to retrieve RDF statements.
     *
     * @return a {@link Statements} instance
     */
    Statements getStatements();

    /**
     * Returns the transaction ID of the current transaction or 0 if no explicit transaction is
     * ↪ available.
     */
}
```

```
*
* @return the transaction ID
*/
long getTransactionId();

/**
 * Returns the update testing status. In a multi-node GraphDB configuration (currently only
↳GraphDB EE) an update
 * will be sent to multiple nodes. The first node that receives the update will be used to test
↳if the update is
 * successful and only if so, it will be send to other nodes. Plugins may use the update test
↳status to perform
 * certain operations only when the update is tested (e.g. indexing data via an external
↳service). The method will
 * return true if this is a GraphDB EE worker node testing the update or this is GraphDB Free or
↳SE. The method will
 * return false only if this is a GraphDB EE worker node that is receiving a copy of the
↳original update
 * (after successful testing on another node).
 *
 * @return true if this update is sent for the first time (testing the update), false otherwise
*/
boolean isTesting();

/**
 * Returns an instance of {@link SystemProperties} that can be used to retrieve various
↳properties that identify
 * the current GraphDB installation and repository.
 *
 * @return an instance of {@link SystemProperties}
*/
SystemProperties getProperties();

/**
 * Returns the repository fingerprint. Note that during an active transactions the fingerprint
↳will be updated
 * at the very end of the transaction. Call it in {@link com.ontotext.trree.sdk.
↳PluginTransactionListener#transactionCompleted(PluginConnection)}
 * if you want to get the updated fingerprint for the just-completed transaction.
 *
 * @return the repository fingerprint
*/
String getFingerprint();

/**
 * Returns the "worker attached" status for the repository. A GraphDB EE worker repository is
↳attached when it is
 * connected to a master. GraphDB SE and Free repository are never attached. This is useful in
↳cases where
 * a plugin may modify the fingerprint via a query. To protect cluster integrity the fingerprint
↳may be changed
 * only via an update.
 *
 * @return true if the repository is a worker repository and is attached to a master, false
↳otherwise
*/
boolean isWorkerAttached();

/*
 * Creates a thread-safe instance of this {@link PluginConnection} that can be used by other
↳threads.
 * Note that every {@link ThreadsafePluginConnecton} must be explicitly closed when no longer
↳needed.
```

```

*
* @return an instance of {@link ThreadsafePluginConnection}
*/
ThreadsafePluginConnection getThreadsafeConnection();
}

```

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 and literals). 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:

```
// 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`:

```
/**
 * This interface represents various properties for the running GraphDB instance and the repository.
 * as seen by the Plugin API.
 */
public interface SystemProperties {
    /**
     * Returns the read-only status of the current repository.
     *
     * @return true if read-only, false otherwise
     */
    boolean isReadOnly();

    /**
     * Returns the number of bits needed to represent an entity id
     *
     * @return the number of bits as an integer
     */
    int getEntityIdSize();
}
```

```

/**
 * Returns the type of the current repository.
 *
 * @return one of {@link RepositoryType#FREE}, {@link RepositoryType#SE} or {@link RepositoryType
↪#EE}
 */
RepositoryType getRepositoryType();

/**
 * Returns the full GraphDB version string.
 *
 * @return a string describing the GraphDB version
 */
String getVersion();

/**
 * Returns the GraphDB major version component.
 *
 * @return the major version as an integer
 */
int getVersionMajor();

/**
 * Returns the GraphDB minor version component.
 *
 * @return the minor version as an integer
 */
int getVersionMinor();

/**
 * Returns the GraphDB patch version component.
 *
 * @return the patch version as an integer
 */
int getVersionPatch();

/**
 * Returns the number of cores in the currently set license up to the physical number of cores.
↪on the machine.
 *
 * @return the number of cores as an integer 0
 */
int getNumberOfLicensedCores();

/**
 * The possible editions for GraphDB repositories.
 */
enum RepositoryType {
    /**
     * GraphDB Free repository
     */
    FREE,
    /**
     * GraphDB SE repository
     */
    SE,
    /**
     * GraphDB EE worker repository
     */
    EE
}
}

```

Repository properties

There are some dynamic repository properties that may change once a repository has been initialized. 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`.

5.8.1.5 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:

```
/**
 * Interface that should be implemented by all plugins that need to maintain per-query context.
 */
public interface Preprocessor {
    /**
     * Pre-processing method called once for every SPARQL query or getStatements() request before it
     ↪is
     * processed.
     *
     * @param request request object
     * @return context object that will be passed to all other plugin methods in the future stages.
     ↪of the
     * request processing
     */
    RequestContext preprocess(Request request);
}
```

The `preprocess(Request request)` method receives the request object and returns a `RequestContext` instance. The passed request parameter is an instance of one of 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, 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:

```
SELECT * WHERE {
  ?s <http://example.com/predicate> ?o
}
```

There is just one statement pattern inside this query: `?s <http://example.com/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:

```
/**
 * Interface implemented by plugins that want to interpret basic triple patterns
 */
public interface PatternInterpreter {
    /**
     * Estimate the number of results that could be returned by the plugin for the given parameters
     *
     * @param subject      subject ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param predicate    predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param object       object ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param context      context value (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext context object as returned by {@code Preprocessor.preprocess()} or
    ↪null
     * @return approximate number of results that could potentially be returned for this parameters
    ↪by the
     * interpret() method
     */
    double estimate(long subject, long predicate, long object, long context, PluginConnection
    ↪pluginConnection,
                    RequestContext requestContext);

    /**
     * Interpret basic triple pattern and return {@link StatementIterator} with results
     *
     * @param subject      subject ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param predicate    predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param object       object ID (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param context      context value (alternatively {@link Entities#BOUND} or {@link Entities
    ↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext context object as returned by {@code Preprocessor.preprocess()} or null
     * @return statement iterator of results
     */
    StatementIterator interpret(long subject, long predicate, long object, long context,
                               PluginConnection pluginConnection, RequestContext requestContext);
}
```

```
}

```

The `estimate()` and `interpret()` methods take the same arguments and are used in the following way:

- Given a statement pattern (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 optimiser 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 optimiser will make an efficient optimisation. 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 optimised, the `interpret()` method of the plugin might be called again should any variable become bound due to the pattern reordering applied by the optimiser. Plugins must be prepared to expect different combinations of bound and unbound statement pattern components, and return appropriate iterators.

The `requestContext` parameter is the value returned by the `preprocess()` method if one exists, or null otherwise.

Results are *returned as statements*.

The plugin framework also supports the interpretation of an extended type of a *list* pattern.

Consider the following SPARQL queries:

```
SELECT * WHERE {
  ?s <http://example.com/predicate> (?o1 ?o2)
}
```

```
SELECT * WHERE {
  (?s1, ?s2) <http://example.com/predicate> ?o
}
```

Internally the object or subject list will be converted to a series of triples conforming to `rdf:List`. These triples can be handled with `PatternInterpreter` but the whole list semantics will have to be implemented by the plugin.

In order to make this task easier the Plugin API defines two additional interfaces very similar to the `PatternInterpreter` interface – `ListPatternInterpreter` and `SubjectListPatternInterpreter`.

`ListPatternInterpreter` handles lists in the *object* position:

```
/**
 * Interface implemented by plugins that want to interpret list-like triple patterns
 */
public interface ListPatternInterpreter {
    /**
     * Estimate the number of results that could be returned by the plugin for the given parameters
     */
}
```



```

    *
    * @param subject          subject ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param objects          object IDs (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param pluginConnection an instance of {@link PluginConnection}
    * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or
↪null
    * @return approximate number of results that could potentially be returned for this parameters
↪by the
    * interpret() method
    */
    double estimate(long subject, long predicate, long[] objects, long context, PluginConnection
↪pluginConnection,
                    RequestContext requestContext);

    /**
    * Interpret list-like triple pattern and return {@link StatementIterator} with results
    *
    * @param subject          subject ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param objects          object IDs (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param pluginConnection an instance of {@link PluginConnection}
    * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or
↪null
    * @return statement iterator of results
    */
    StatementIterator interpret(long subject, long predicate, long[] objects, long context,
                               PluginConnection pluginConnection, RequestContext requestContext);
}

```

It differs from `PatternInterpreter` by having multiple objects passed as an array of `long`, instead of a single `long` object. The semantics of both methods is equivalent to the one in the basic pattern interpretation case.

`SubjectListPatternInterpreter` handles lists in the *subject* position:

```

/**
 * Interface implemented by plugins that want to interpret list-like triple patterns
 */
public interface SubjectListPatternInterpreter {
    /**
    * Estimate the number of results that could be returned by the plugin for the given parameters
    *
    * @param subjects          subject IDs (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param predicate        predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param object           object ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param context          context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
    * @param pluginConnection an instance of {@link PluginConnection}
    * @param requestContext   context object as returned by {@code Preprocessor.preprocess()} or
↪null
    */
}

```

```

    * @return approximate number of results that could potentially be returned for this parameters.
↪by the
    * interpret() method
    */
    double estimate(long[] subjects, long predicate, long object, long context, PluginConnection
↪pluginConnection,
        RequestContext requestContext);

    /**
     * Interpret list-like triple pattern and return {@link StatementIterator} with results
     *
     * @param subjects      subject IDs (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param predicate     predicate ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param object        object ID (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param context       context value (alternatively {@link Entities#BOUND} or {@link Entities
↪#UNBOUND})
     * @param pluginConnection an instance of {@link PluginConnection}
     * @param requestContext context object as returned by {@code Preprocessor.preprocess()} or null
     * @return statement iterator of results
     */
    StatementIterator interpret(long[] subjects, long predicate, long object, long context,
        PluginConnection pluginConnection, RequestContext requestContext);
}

```

It differs from `PatternInterpreter` by having multiple subjects passed as an array of `long`, instead of a single `long` subject. The semantics of both methods 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:

```

/**
 * 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 context object as returned by {@link Preprocessor#preprocess(Request)}
↪ (in case this plugin

```

```

    *                                     implemented this interface)
    * @return binding set object that should be post-processed further by next post-processing_
↪ plugins or
    * null if the current binding set should be deleted from the result set
    */
    BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext);

    /**
    * Method called after all post-processing has been finished for each plugin. This is the point_
↪ where
    * every plugin could introduce its results even if the original result set was empty
    *
    * @param requestContext context objected as returned by {@link Preprocessor#preprocess(Request)}
↪ (in case this plugin
    *                                     implemented this interface)
    * @return iterator for resulting binding sets that need to be added to the final result set
    */
    Iterator<BindingSet> flush(RequestContext requestContext);
}

```

The `postprocess()` method is called for each binding set that is to be returned to the repository client. This method may modify the binding set and return it, or alternatively, return `null`, in which case the binding set is removed from the result set. After a binding set is processed by a plugin, the possibly modified binding set is passed to the next plugin 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.

5.8.1.6 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 initialisation, 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.

```

/**
 * An interface that should be implemented by the plugins that want to be notified for particular_
↪ update
 * events. The getPredicatesToListenFor() method should return the predicates of interest to the_
↪ plugin. This
 * method will be called once only immediately after the plugin has been initialized. After that_
↪ point the
 * plugin's interpretUpdate() method will be called for each inserted or deleted statement sharing_
↪ one of the
 * predicates of interest to the plugin (those returned by getPredicatesToListenFor()).
 */
public interface UpdateInterpreter {
    /**
    * Returns the predicates for which the plugin needs to get notified when a statement with such_
↪ a predicate is added or removed.
    *
    * @return array of predicates as entity IDs
    */
}

```

```

long[] getPredicatesToListenFor();

/**
 * Hook that is called whenever a statement containing one of the registered predicates
 * (see {@link #getPredicatesToListenFor()}) is added or removed.
 *
 * @param subject      subject value of the updated statement
 * @param predicate     predicate value of the updated statement
 * @param object        object value of the updated statement
 * @param context       context value of the updated statement
 * @param isAddition    true if the statement was added, false if it was removed
 * @param isExplicit    true if the updated statement was explicit one
 * @param pluginConnection an instance of {@link PluginConnection}
 * @return true - when the statement was handled by the plugin only and should <i>NOT</i> be
↳ added to/removed from the repository,
 * false - when the statement should be added to/removed from the repository
 */
boolean interpretUpdate(long subject, long predicate, long object, long context, boolean
↳ isAddition,
                        boolean isExplicit, PluginConnection pluginConnection);
}

```

Removal of entire contexts

Statement deletion in GraphDB is specified as a quadruple (*subject, predicate, object, context*), where each position can be explicit or null. Null in this case means all subjects, predicates, objects or contexts depending on the position where null was specified.

When at least one of the positions is non-null, the plugin framework will fire individual events for each matching and removed statement.

When all positions are null (i.e. delete everything in the repository) the operation will be optimized internally and individual events will not be fired. This means that `UpdateInterpreter` and `StatementListener` will not be called.

`ClearInterpreter` is an interface that allows plugins to detect the removal of entire contexts or removal of all data in the repository:

```

/**
 * This interface can be implemented by plugins that want to be notified on clear()
 * or remove() (all statements in any context).
 */
public interface ClearInterpreter {
    /**
     * Notification called before the statements are removed from the given context.
     *
     * @param context      the ID of the context or 0 if all contexts
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void beforeClear(long context, PluginConnection pluginConnection);

    /**
     * Notification called after the statements have been removed from the given context.
     *
     * @param context      the ID of the context or 0 if all contexts
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void afterClear(long context, PluginConnection pluginConnection);
}

```

Intercepting data for specific contexts

The Plugin API provides a way to intercept data inserted into or removed from a particular predefined context. The ContextUpdateHandler interface:

```
/**
 * This interface provides a mechanism for plugins to handle updates to certain contexts.
 * When a plugin requests handling of a context, all data for that context will be 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.

5.8.1.7 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:

```
/**
 * The {@link PluginTransactionListener} allows plugins to be notified about transactions (start,
 * commit+completed or abort)
 */
public interface PluginTransactionListener {
    /**
     * Notifies the listener about the start of a transaction.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionStarted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the commit phase of a transaction. Plugins should use this event
     * to perform their own
     * commit work if needed or to abort the transaction if needed.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionCommit(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the completion of a transaction. This will be the last event in a
     * successful transaction.
     * The plugin is not allowed to throw any exceptions here and if so they will be ignored. If a
     * plugin needs to abort
     * a transaction it should be done in {@link #transactionCommit(PluginConnection)}.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionCompleted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about the abortion of a transaction. This will be the last event in an
     * aborted transaction.
     * <p>
     * Plugins should revert any modifications caused by this transaction, including the fingerprint.
     *
     * @param pluginConnection an instance of {@link PluginConnection}
     */
    void transactionAborted(PluginConnection pluginConnection);

    /**
     * Notifies the listener about a user abort request. A user abort request is a request by an end-
     * user to abort the
     * transaction. Unlike the other events this will be called asynchronously whenever the request
     * is received.
     * <p>
     * Plugins may react and terminate any long-running computation or ignore the request. This is
     * just a handy way
     * to speed up abortion when a user requests it. For example, this event may be received
     * asynchronously while
     * the plugin is indexing data (in {@link #transactionCommit(PluginConnection)} running in the
     * main thread).
     * The plugin may notify itself that the indexing should stop. Regardless of the actions taken
     * by the plugin

```

```

    * the transaction may still be aborted and {@link #transactionAborted(PluginConnection)} will
    ↪be called.
    * All clean up of the abortion should be handled in {@link
    ↪#transactionAborted(PluginConnection)}.
    *
    * @param pluginConnection an instance of {@link PluginConnection}
    */
    default void transactionAbortedByUser(PluginConnection pluginConnection) {

    }
}

```

Each transaction has a beginning signalled by a call to `transactionStarted()`. Then the transaction can proceed in several ways:

- Commit and completion:
 - `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.

5.8.1.8 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.

5.8.1.9 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 `RDFRank` 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 `PluginDependency` interface:

```
/**
 * Interface that should be implemented by plugins that depend on other plugins and want to be able to
 * retrieve references to them at runtime.
 */
public interface PluginDependency {
    /**
     * Method used by the plugin framework to inject a {@link PluginLocator} instance in the plugin.
     *
     * @param locator a {@link PluginLocator} instance
     */
    void setLocator(PluginLocator locator);
}
```

They are then injected into an instance of the PluginLocator interface (during the configuration phase), which does the actual plugin discovery for them:

```
/**
 * Interface that supports obtaining of a plugin instance by plugin name. An object implementing this
 * interface is injected into plugins that implement the {@link PluginDependency} interface.
 */
public interface PluginLocator {
    /**
     * Retrieves a {@link Plugin} instance by plugin name
     *
     * @param name name of the plugin
     * @return a {@link Plugin} instance or null if a plugin with that name is not available
     */
    Plugin locate(String name);

    /**
     * Retrieves a {@link RDFRankProvider} instance.
     *
     * @return a {@link RDFRankProvider} instance or null if no {@link RDFRankProvider} is available
     */
    RDFRankProvider locateRDFRankProvider();
}
```

Having a reference to another plugin is all that is needed to call its methods directly and make use of its services.

An important interface related to accessing other plugins is the RDFRankProvider interface. The sole implementation is the RDFRank 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.

5.8.1.10 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 and literals).

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`.

5.8.1.11 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:

```
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:

```
public class ExamplePlugin extends PluginBase implements PatternInterpreter {

    // ... initialize() and getName()

    // PatternInterpreter interface methods
    @Override
    public StatementIterator interpret(long subject, long predicate, long object, long context,
                                     PluginConnection pluginConnection, RequestContext requestContext) {
        // Ignore patterns with predicate different than the one we are interested in. We want to
        // return the
        // SystemDate only when we detect the <http://example.com/time> predicate.
        if (predicate != predicateId)
            // This will tell the PluginManager that we can't interpret the statement so the
            // statement can be passed
            // to another plugin.
            return null;

        // Create the date/time literal. Here it is important to create the literal in the entities
        // instance of the
        // request and NOT in getEntities(). If you create it in the entities instance returned by
        // getEntities() it
        // will not be visible in the current request.
        long literalId = createDateTimeLiteral(pluginConnection.getEntities());

        // return a StatementIterator with a single statement to be iterated. The object of this
        // statement will be the
        // current timestamp.
        return StatementIterator.create(subject, predicate, literalId, 0);
    }

    @Override
    public double estimate(long subject, long predicate, long object, long context,
                          PluginConnection pluginConnection, RequestContext requestContext) {
        // We always return a single statement so we return a constant 1. This value will be used by
        // the QueryOptimizer
        // when crating the execution plan.
        return 1;
    }

    private long createDateTimeLiteral(Entities entities) {
        // Create a literal for the current timestamp.
        Value literal = SimpleValueFactory.getInstance().createLiteral(new Date());

        // Add the literal in the entity pool with REQUEST scope. This will make the literal
        // accessible only for the
        // current Request and will be disposed once the request is completed. Return it's ID.
        return entities.put(literal, Entities.Scope.REQUEST);
    }
}
```

The interpret() method only processes patterns with a predicate matching the desired predicate identifier. Further on, it simply creates a new date/time literal (in the request scope) and places its identifier in the object position

of the returned single result. The `estimate()` method always returns 1, because this is the exact size of the result set.

ExamplePlugin

ExamplePlugin 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:

```
public class ExamplePlugin extends PluginBase implements Preprocessor, Postprocessor {
    // ... initialize() and getName()

    // Preprocessor interface methods
    @Override
    public RequestContext preprocess(Request request) {
        // We are interested only in QueryRequests
        if (request instanceof QueryRequest) {
            QueryRequest queryRequest = (QueryRequest) request;
            Dataset dataset = queryRequest.getDataset();

            // Check if the predicate is included in the default graph. This means that we have a
            ↪ "FROM <our_predicate>"
            // clause in the SPARQL query.
            if ((dataset != null && dataset.getDefaultGraphs().contains(timeIri))) {
                // Create a date/time literal
                Value literal = createDateTimeLiteral();

                // Prepare a binding set with all projected variables set to the date/time literal.
            ↪ value
                MapBindingSet result = new MapBindingSet();
                for (String bindingName : queryRequest.getTupleExpr().getBindingNames()) {
                    result.addBinding(bindingName, literal);
                }

                // Create a Context object which will be available during the other phases of the
            ↪ request processing
                // and set the created result as an attribute.
                RequestContextImpl context = new RequestContextImpl();
                context.setAttribute("bindings", result);

                return context;
            }
            // If we are not interested in the request there is no need to create a Context.
            return null;
        }

        // Postprocessor interface methods
        @Override
        public boolean shouldPostprocess(RequestContext requestContext) {
            // Postprocess only if we have created RequestContext in the Preprocess phase. Here the
            ↪ requestContext object
            // is the same one that we created in the preprocess(...) method.
            return requestContext != null;
        }

        @Override
        public BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext) {
            // Filter all results. Returning null will remove the binding set from the returned query.
            ↪ result.
            // We will add the result we want in the flush() phase.
            return null;
        }
    }
}
```



```

@Override
public Iterator<BindingSet> flush(RequestContext requestContext) {
    // Get the BindingSet we created in the Preprocess phase and return it.
    // This will be returned as the query result.
    BindingSet result = (BindingSet) ((RequestContextImpl) requestContext).getAttribute("bindings");
    return new SingletonIterator<>(result);
}

private Literal createDateTimeLiteral() {
    // Create a literal for the current timestamp.
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.HOUR, timeOffsetHrs);

    return SimpleValueFactory.getInstance().createLiteral(calendar.getTime());
}
}

```

The plugin creates an instance of RequestContext using the default implementation RequestContextImpl. It can hold attributes of any type referenced by a name. Then the plugin creates a BindingSet with the date/time literal, bound to every variable name in the query projection, and sets it as an attribute with the name “bindings”. The postprocess() method filters out all results if the requestContext is non-null (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:

```

public class ExamplePlugin extends PluginBase implements UpdateInterpreter, Preprocessor,
Postprocessor {
    // ... initialize() and getName()

    // ... Pre- and Postprocessor methods

    // UpdateInterpreter interface methods
    @Override
    public long[] getPredicatesToListenFor() {
        // We can filter the tuples we are interested in by their predicate. We are interested only
        // in tuples with have the predicate we are listening for.
        return new long[] {goFutureID, goPastID};
    }

    @Override
    public boolean interpretUpdate(long subject, long predicate, long object, long context, boolean
isAddition,
                                boolean isExplicit, PluginConnection pluginConnection) {
        // Make sure that the subject is the time entity
        if (subject == timeID) {
            final String intString = pluginConnection.getEntities().get(object).stringValue();
            int step;
            try {
                step = Integer.parseInt(intString);
            } catch (NumberFormatException e) {
                // Invalid input, propagate the error to the caller
                throw new ClientErrorException("Invalid integer value: " + intString);
            }

            if (predicate == goFutureID) {
                timeOffsetHrs += step;
            } else if (predicate == goPastID) {
                timeOffsetHrs -= step;
            }
        }
    }
}

```

```
        // We handled the statement.
        // Return true so the statement will not be interpreted by other plugins or inserted in_
↪the DB
        return true;
    }

    // Tell the PluginManager that we can not interpret the tuple so further processing can_
↪continue.
    return false;
}
}
```

UpdateInterpreter must specify the predicates 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.

5.8.2 RDF rank

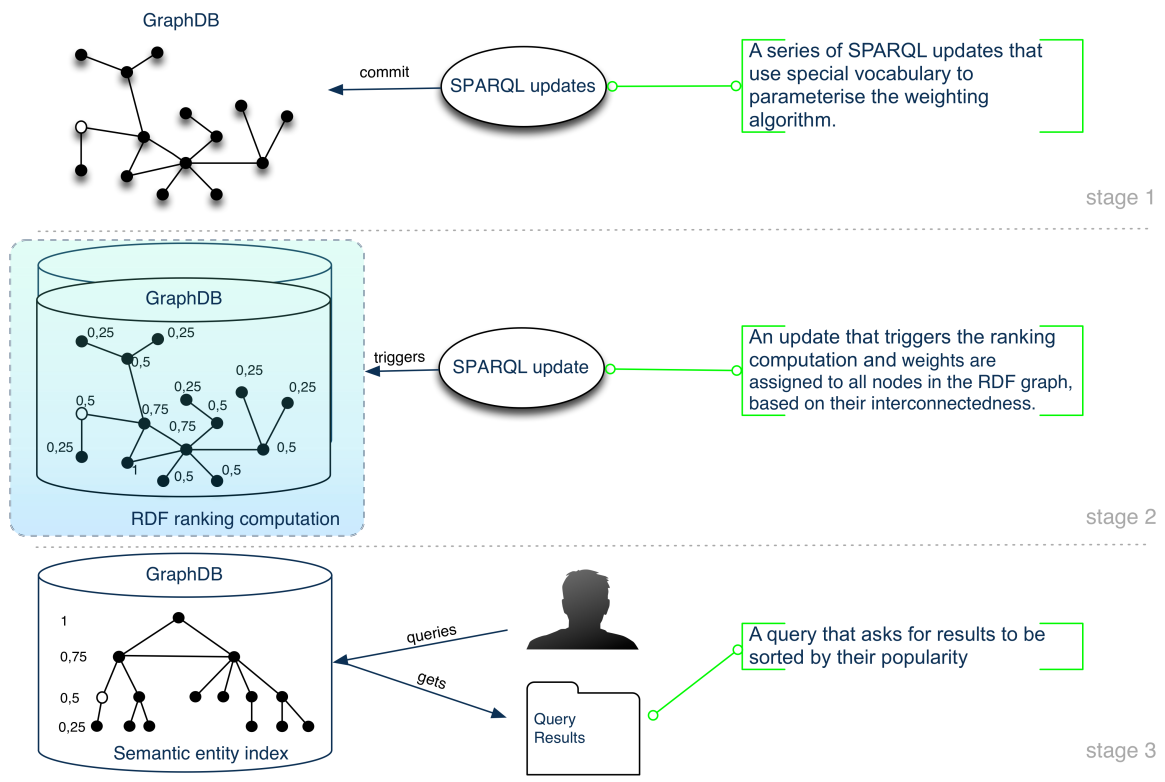
What's in this document?

- *What is RDF Rank*
- *Parameters*
- *Full computation*
- *Incremental updates*
- *Exporting RDF Rank values*
- *Checking the RDF Rank status*
- *Rank Filtering*

5.8.2.1 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 and literals. 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#>
PREFIX opencyc-en: <http://sw.opencyc.org/2008/06/10/concept/en/>
SELECT * WHERE {
  ?Person a opencyc-en:Entertainer .
  ?Person rank:hasRDFRank ?rank .
}
ORDER BY DESC(?rank) LIMIT 100
```

As 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 parameterise the weighting algorithm, followed by an update that triggers the computation itself.

5.8.2.2 Parameters

RDF Rank is fully controllable from *Setup* -> *RDF Rank*.

Parameter	Maximum iterations
Predicate	<code>http://www.ontotext.com/owlim/RDFRank#maxIterations</code>
Description	Sets the maximum number of iterations of the algorithm over all entities in the repository.
Default	20
Example	PREFIX rank: < <code>http://www.ontotext.com/owlim/RDFRank#</code> > INSERT DATA { rank:maxIterations rank:setParam “16” . }

Parameter	Epsilon
Predicate	<code>http://www.ontotext.com/owlim/RDFRank#epsilon</code>
Description	Terminates the weighting algorithm early when the total change of all RDF Rank scores has fallen below this value.
Default	0.01
Example	PREFIX rank: < <code>http://www.ontotext.com/owlim/RDFRank#</code> > INSERT DATA { rank:epsilon rank:setParam “0.05” . }

5.8.2.3 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 the *Compute Full* button.

RDF Rank ⓘ

RDFRank for repository wine is with status ⓘ RDFRank not built yet

Compute Full

Filtering OFF ☐

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*.

5.8.2.4 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.

5.8.2.5 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.

5.8.2.6 Checking the RDF Rank status

The RDF Rank plugin can be in one of the following statuses:

```
/**
 * The ranks computation has been canceled
 */
CANCELED,

/**
 * The ranks are computed and up-to-date
 */
COMPUTED,
```

```
/**
 * A computing task is currently in progress
 */
COMPUTING,

/**
 * There are no calculated ranks
 */
EMPTY,

/**
 * Exception has been thrown during computation
 */
ERROR,

/**
 * The ranks are outdated and need computing
 */
OUTDATED,

/**
 * The filtering is enabled and it's configuration has been changed since the last full_
↪computation
 */
CONFIG_CHANGED
```

You can get the current status of the plugin by running the following query:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?o WHERE { ?s rank:status ?o }
```

5.8.2.7 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:

```
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. Below you can find how to control the lists with SPARQL queries:

Get the content of a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
SELECT ?s WHERE { ?s rank:includedPredicates ?o }
```

Add an IRI to a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { <http:predicate> rank:includedPredicates "add" }
```

Remove an IRI from a list:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { <http:predicate> rank:includedPredicates "remove" }
```

The filtering can be done not only by including statements of interest but by removing ones as well. In order to do so there are 2 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 2 parameters - *includeExplicit* and *includeImplicit* which are set to *true* by default. When these parameters are set to *true* they are just disregarded - 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:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:includeExplicit _:b2 . }
```

You can set value of the parameters with:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
INSERT DATA { rank:includeExplicit rank:setParam true }
```

5.8.3 Semantic similarity searches

What's in this document?

- *Why do I need the similarity plugin?*
- *What the similarity plugin does?*
- *How the similarity plugin works?*
 - *Search similar terms*
 - *Search documents for which selected term is specific*
 - *Search specific terms in selected document*
 - *Search for closest documents*
- *Download data*
- *Text based similarity searches*
 - *Create text similarity index*
 - *Create index parameters*
 - *Disabled parameters*
 - *Stop words and Lucene Analyzer*
 - *Search in the index*
 - *Search parameters*
 - *Locality-sensitive hashing*
 - *Search similar news within days*
 - *Term to term search*
 - *Boosting a term's weight*

- *Predication-based Semantic Indexing*
 - *Load example data*
 - *Create predication-based index*
 - *Search predication-based index*
 - *Analogical searches*
 - *Why is this important?*
- *Hybrid indexing*
 - *Edit the FactForge data*
 - *Create a Literal index*
 - *Use the Literal index*
 - *Indexing behavior*
 - *Manual creation*
- *Training cycles*

5.8.3.1 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.

5.8.3.2 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

5.8.3.3 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.

Search in News

Search RDF resources or enter term keywords

search options 

Search type: ☒ Term ☐ Document

Result type: ☒ Term ☐ Document

Semantic Vectors search parameters:

-numsearchresults 20 -searchtype mir

[See the full list of supported parameters](#)

Showing results for "novichok" [View SPARQL Query](#)

	documentID	score
1	novichok	"1.0""xsd:double
2	porton	"0.5792213617185374""xsd:double
3	salisbury	"0.5393066920495809""xsd:double
4	nerve	"0.5371825357626078""xsd:double
5	skripal	"0.4806951938822142""xsd:double
6	skripals	"0.46424357744188954""xsd:double
7	mirzayanov	"0.4462146390828783""xsd:double
8	yulia	"0.4394177895811721""xsd:double
9	poisoning	"0.4359686538131075""xsd:double
10	agent	"0.43257349963137026""xsd:double

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.

Search in News

Search RDF resources or enter term keywords

search options 

Search type: ☒ Term ☐ Document

Result type: ☐ Term ☒ Document

Semantic Vectors search parameters:

`-numsearchresults 20 -searchtype mir`

[See the full list of supported parameters](#)

Showing results for "novichok" [View SPARQL Query](#)

	documentID 	
1	http://www.bbc.co.uk/news/uk-43876669	"0.57
2	https://www.theguardian.com/politics/2018/apr/15/boris-johnson-defends-remarks-on-russian-role-in-novichok-case	"0.55
3	https://www.theguardian.com/uk-news/2018/apr/12/novichok-used-in-spy-poisoning-chemical-weapons-watchdog-confirms-salisbury	"0.54
4	https://www.9news.com.au/world/2018/04/18/07/00/nerve-agent-that-poisoned-spy-was-in-liquid-form	"0.54
5	http://www.businessinsider.com/skripal-poisoning-nerve-agent-delivered-in-liquid-form-uk-says-2018-4	"0.53
6	https://www.theguardian.com/world/2018/apr/06/uk-us-case-file-russian-nerve-agent-shikhany-spy-poisoning	"0.52
7	https://www.reuters.com/article/us-britain-russia-explainer/what-is-known-and-not-known-about-poisoning-of-ex-spy-in-britain-idUSKCN1HC148	"0.51
8	https://sputniknews.com/europe/201803301063066281-bulgaria-refrain-expelling-russian-diplomats/	"0.51
9	http://www.bbc.com/news/uk-43798068	"0.51
10	http://www.scmp.com/news/world/europe/article/2140173/uk-lab-says-cannot-determine-	"0.51

Search specific terms in selected document

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 in News

Search RDF resources or enter term keywords

search options ▼

Search type: ☐ Term ☒ Document

Result type: ☒ Term ☐ Document

Semantic Vectors search parameters:

-numsearchresults 20 -searchtype mir

[See the full list of supported parameters](#)

Showing results for <http://www.bbc.co.uk/news/uk-43876669> [View SPARQL Query](#)

	documentID	score
1	novichok	"0.5703498341722145"^^xsd:double
2	poisoning	"0.5520181996943301"^^xsd:double
3	nerve	"0.5370584264445704"^^xsd:double
4	salisbury	"0.5264277793825664"^^xsd:double
5	skripal	"0.5249264138661435"^^xsd:double

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.

Search in News

Search RDF resources or enter term keywords

search options 

Search type: ☐ Term ☒ Document

Result type: ☐ Term ☒ Document

Semantic Vectors search parameters:

`-numsearchresults 20 -searchtype mir`

[See the full list of supported parameters](#)

Showing results for <http://www.bbc.co.uk/news/uk-43876669> [View SPARQL Query](#)

	documentID 	
1	http://www.bbc.co.uk/news/uk-43876669	"1.0"
2	https://www.theglobeandmail.com/world/article-nerve-agent-used-to-poison-sergei-skripal-was-in-liquid-form/	"0.8"
3	https://www.theguardian.com/uk-news/2018/mar/29/yulia-skripal-no-longer-in-critical-condition-says-salisbury-nhs	"0.8"
4	https://www.irishtimes.com/news/world/uk/yulia-skripal-daughter-of-poisoned-russian-spy-improving-rapidly-1.3444674	"0.8"
5	https://www.9news.com.au/world/2018/04/18/07/00/nerve-agent-that-poisoned-spy-was-in-liquid-form	"0.8"
6	https://www.reuters.com/article/us-britain-russia-explainer/what-is-known-and-not-known-about-poisoning-of-ex-spy-in-britain-idUSKCN1HC148	"0.8"
7	https://www.expressandstar.com/news/uk-news/2018/03/31/russia-claims-british-officials-searched-aeroflot-plane-at-heathrow/	"0.8"
8	https://www.independent.co.uk/news/uk/crime/yulia-skripal-latest-nerve-agent-critical-condition-salisbury-poisoning-russia-spy-a8279761.html	"0.8"
9	https://www.theguardian.com/uk-news/2018/mar/27/government-approves-1m-recovery-package-salisbury	"0.8"
10	https://www.independent.ie/irish-news/deadly-nerve-agent-in-highest-concentration-on-exspys-front-door-tests-reveal-36754247.html	"0.8"

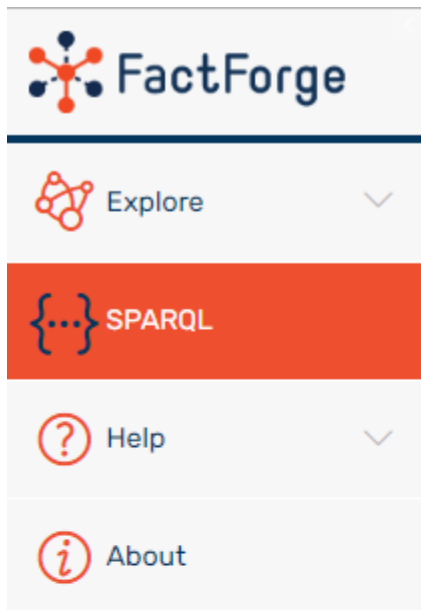
5.8.3.4 Download data

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> 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 ff-map:mentionsEntity ?entity .
    ?document pubo:content ?content .
    ?document pubo:creationDate ?date .
} WHERE {
    ?document a pubo:Document .
    ?document ff-map:mentionsEntity ?entity .
    ?document pubo:content ?content .
    ?document pubo:creationDate ?date .
    FILTER (?p NOT IN (pubo:containsMention, pubo:hasFeature, pubo:hasImage))
    FILTER ( (?date > "2018-01-01"^^xsd:dateTime) && (?date < "2018-04-30"^^xsd:dateTime))
}
```



SPARQL Query & Update ?

Unnamed × ⊕

```

1  PREFIX pubo: <http://ontology.ontotext.com/publishing/
2  PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
3  PREFIX dbr: <http://dbpedia.org/resource/>
4  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5  PREFIX ff-map: <http://factforge.net/ff2016-mapping/>
6
7  CONSTRUCT {
8      ?document ff-map:mentionsEntity ?entity .
9      ?document pubo:content ?content .
10     ?document pubo:creationDate ?date .
11  } WHERE {
12     {
13         ?document a pubo:Document .
14         ?document ff-map:mentionsEntity ?entity .
15         ?document pubo:content ?content .
16         ?document pubo:creationDate ?date .

```

Table

Raw Response

Pivot Table

Google Chart

Filter query results

Showing results fr

Download the data using the *Download As* button - choose 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->RDF->Server* files (see here how to *import server files*).

Import the `query-result.ttl` file in your “news” repository.

Go to *Setup*, enable *Autocomplete index*, and create an index for *allNews*, using *Build Now* button.

The *Autocomplete index* is used for autocompletion of URLs in the SPARQL editor and the View resource page.

Autocomplete index

Autocomplete for repository news is **ON**  with status **Ready** 

5.8.3.5 Text based similarity searches

Create text similarity index

Create index for allNews in the following way:

Go to *Explore* -> *Similarity* -> *Create similarity index* and change the search query to:

```
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
SELECT ?documentID ?documentText
{
  ?documentID pubo:content ?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

Select one of your existing indexes to search in it

allNews

Text

Create index parameters

- **-vectortype** - real, complex, binary - 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's 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's 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 stoplist.
- **-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>> predicate. Stop words are passed as a string literal as opposed to a file.
- **-elementalmethod**
- **-docindexing**

Stop words and Lucene Analyzer

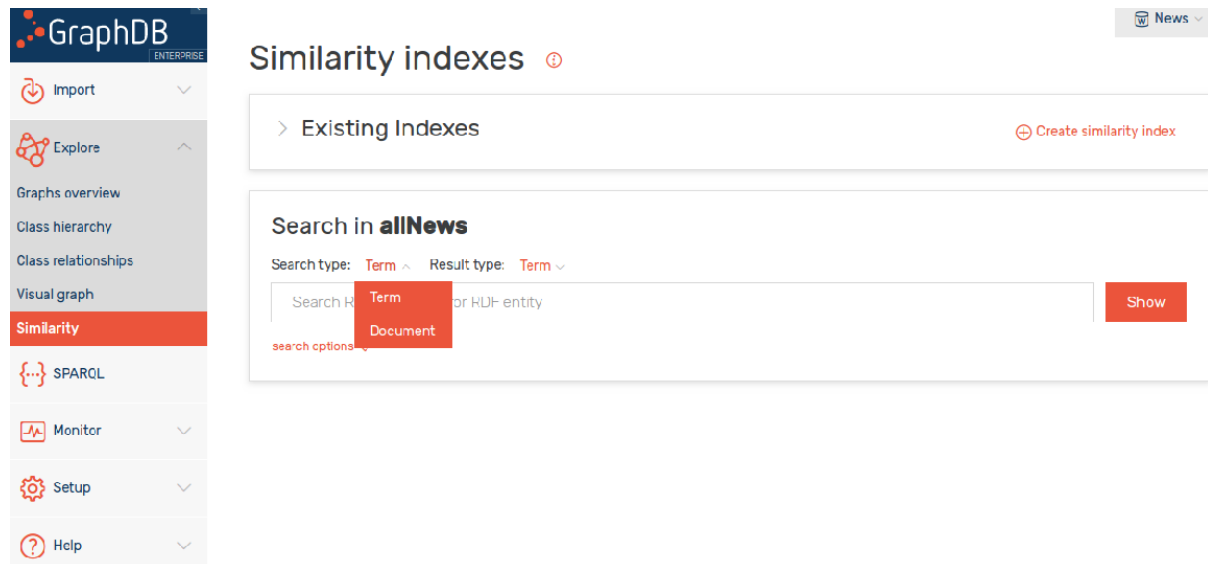
In the *Stop words* field, add a custom list of stop words to be passed to the Semantic Vector plugin. If left empty, the default Lucene stop words list will be used.

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 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*.



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 is false.
- **-numsearchresults** - number of search results.

Additional info at: <https://github.com/semanticvectors/semanticvectors/wiki/SearchOptions>

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:

- A search vector is generated
- 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 n random vectors used for hashing. Default value: 0.
- **-lsh_max_bits_diff** - the m number of bits by which two hashes can differ and still be considered similar. Default value: 0.

The hashing workflow is as follows:

- A n number of random orthogonal vectors are generated
- Each vector in store is compared to each of those vectors (checking whether their scalar product is positive or not)
- Given this data, a hash is generated for each of the vectors in store

During a search, the workflow is:

- A search vector is generated
- A hash is generated for this search vector by comparing it to the n number of random vectors used during the initial hashing
- 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)
- 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.

Note: If both parameters have the same value, then all possible hashes are considered similar and therefore no filtering is done. For optimization purposes in this scenario, the entire hashing logic has been bypassed.

If one of the parameters is specified during the index creation, then its value will be used as the default one for searching.

Depending on its configuration, the hash can perform in different ways.

A higher 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 n 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 m number of **-lsh_max_bits_diff** with an n number of hashes leads to m -combinations of $n + (m - 1)$ -combination of $n + \dots + 0$ -combinations of n checked buckets.

By modifying these parameters, you can control the number of checked vectors. A lower number of checked vectors leads to higher performance, but also increases the chance of missing a similar vector.

Different settings perform well for different vector store sizes. A reasonable initial configuration is (3, 1). If you want to slightly increase the precision, you could 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:

```
# Search similar news (SPARQL) within days

PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?entity ?score ?matchDate ?searchDate {
  BIND (<http://www.uawire.org/merkel-and-putin-discuss-syria-and-nord-stream-2> as ?
↪searchDocumentID)
  ?search a inst:allNews ;
    :searchDocumentID ?searchDocumentID;
    :searchParameters "";
    :documentResult ?result .
  ?result :value ?entity ;
    :score ?score .
  ?entity pubo:creationDate ?matchDate .
  ?searchDocumentID pubo:creationDate ?searchDate .
  FILTER (?matchDate > ?searchDate - "P2D"^^xsd:duration && ?matchDate < ?searchDate + "P2D"^^
↪xsd:duration)
}
```

Search for similar news, get their creationDate and filter only the news within the time period of 2 days.

Table	Raw Response	Pivot Table	Google Chart	Download as
Filter query results				Showing results from 1 to 9 of 9. Query took 0.5s, moments ago.
	entity	score	matchDate	searchDate
1	https://www.thelocal.de/20180813/merkel-to-meet-putin-on-saturday	"1.0""xsd:double	"2018-08-13T10:41:15Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
2	http://tass.com/politics/1017090	"0.9322463953531154""xsd:double	"2018-08-13T17:48:45Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
3	https://uk.reuters.com/article/uk-germany-russia/merkel-putin-to-meet-outside-berlin-on-aug-18-german-spokesman-idUKKBN1KY0Z5	"0.9314538047209093""xsd:double	"2018-08-13T09:45:00Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
4	http://www.arabnews.com/node/1355616/world	"0.9249599701802667""xsd:double	"2018-08-13T11:03:45Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
5	https://www.express.co.uk/news/world/1003025/Vladimir-Putin-Angela-Merkel-meeting-Germany-Schloss-Meseberg-Ukraine-Crimea-Nord-Stream-2	"0.9208576781305563""xsd:double	"2018-08-14T01:13:28Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
6	http://www.pravdareport.com/news/russia/kremlin/14-08-2018/141397-putin_merkel-0/	"0.9186393927406488""xsd:double	"2018-08-14T15:45:00Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
7	https://www.newsweek.com/what-will-putin-merkel-talk-about-berlin-meeting-1072426	"0.9186272120223427""xsd:double	"2018-08-14T16:18:45Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
8	http://tass.com/world/1017034	"0.9165053678672124""xsd:double	"2018-08-13T10:27:23Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime
9	https://www.nytimes.com/2018/08/13/world/europe/another-surprise-meeting-with-putin-this-time-its-merkel.html	"0.9150726062021857""xsd:double	"2018-08-13T18:11:15Z""xsd:dateTime	"2018-08-13T10:41:15Z""xsd:dateTime

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:

```
PREFIX pubo: <http://ontology.ontotext.com/publishing#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.ontotext.com/graphdb/similarity/>
PREFIX inst: <http://www.ontotext.com/graphdb/similarity/instance/>

SELECT ?documentID ?documentText {
  ?documentID pubo:content ?documentText .
  ?documentID pubo:creationDate ?date .
  FILTER ( (?date > "2018-01-01"^^xsd:dateTime) && (?date < "2018-01-30"^^xsd:dateTime))
  FILTER(isLiteral(?documentText))
}
```

Do the same for February, March, and April by changing the date range. 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.

Show

search options ▾

Showing results for "korea" [View SPARQL Query](#)

	entity	score
1	korea	"1.0000000009913526""xsd:double
2	north	"0.8654814562036028""xsd:double
3	korean	"0.7896469590401117""xsd:double
4	south	"0.7172887559107783""xsd:double
5	korea's	"0.6900821636779971""xsd:double
6	seoul	"0.6618936474102592""xsd:double
7	pyongyang	"0.6279485337216868""xsd:double
8	olympics	"0.6002342474268424""xsd:double
9	kim	"0.5567806922009881""xsd:double
10	jong	"0.5538292080558774""xsd:double
11	nuclear	"0.5422974153347758""xsd:double
12	koreas	"0.5321398975392728""xsd:double

Search in **allNewsApril**

[search options](#) ▼

 Showing results for "korea" [View SPARQL Query](#)

	entity	score
1	korea	"1.0""xsd:double
2	north	"0.8497155647442292""xsd:double
3	korean	"0.7095242083478267""xsd:double
4	korea's	"0.6750772240411883""xsd:double
5	jong	"0.6526829053731862""xsd:double
6	south	"0.6366991195090274""xsd:double
7	kim	"0.6301615258499672""xsd:double
8	jae	"0.5437940444464109""xsd:double
9	peninsula	"0.5379938384807824""xsd:double
10	summit	"0.5363756674103929""xsd:double
11	seoul	"0.49683890581694795""xsd:double
12	denuclearization	"0.48414914645439705""xsd:double
13	koreans	"0.47691733577903833""xsd:double
14	nuclear	"0.4678922565615057""xsd:double

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.

5.8.3.6 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 more than 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*):

- dbo : <http://dbpedia.org/ontology/>
- dbr : <http://dbpedia.org/resource/>
- 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*.

Create similarity index ⓘ

 people-60s ▾

Create text similarity index

Create predication index

Index name

People-60s

more options ^

Semantic Vectors create index parameters

-termweight idf

See the full [list of supported parameters](#)

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 `-termweight idf`. Also, for better results, set `-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:

- ?subject
- ?predicate
- ?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 {
    ?subject ?predicate ?object .
}
```

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 :<http://www.ontotext.com/graphdb/similarity/>
PREFIX inst:<http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX psi:<http://www.ontotext.com/graphdb/similarity/psi/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?entity ?name ?description ?birthDate ?birthPlace ?gender ?score {
    ?search a ?index ;
        ?searchType ?query;
        psi:searchPredicate ?psiPredicate;
        :searchParameters ?parameters;
    ?resultType ?result .
    ?result :value ?entity ;
        :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* button 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:

Edit Search, Analogical queries



Search query

Analogical query

```
PREFIX :<http://www.ontotext.com/graphdb/similarity/>
PREFIX inst:<http://www.ontotext.com/graphdb/similarity/instance/>
PREFIX psi:<http://www.ontotext.com/graphdb/similarity/psi/>

SELECT ?entity ?score {
  ?search a ?index ;
    ?searchType ?query;
    psi:searchPredicate ?psiPredicate;
    :searchParameters ?parameters;
    ?resultType ?result .
  ?result :value ?entity ;
    :score ?score .
}
```

Cancel

Edit Query

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.

people-60s ▾

Similarity indexes ⓘ

Existing Indexes

[⊕ Create similarity index](#)

People-60s

Predication



Search in People-60s

Search type: IRI ▾

Hristo_

Show

http://apeia.org/resource/Hristo_Kolev
http://dbpedia.org/resource/Hristo_Shopov
http://dbpedia.org/resource/Hristo_Koilov
http://dbpedia.org/resource/Hristo_Kovachki
http://dbpedia.org/resource/Hristo_Stoichkov
http://dbpedia.org/resource/Hristo_Kyuchukov
http://dbpedia.org/resource/Hristo_Atanasov
[search options ^](#)

Semantic Vectors search parameters:

In the results, you can see Bulgarian football players born in the same town, other Bulgarian athletes born in the same place, as well as other people with the same birth date.

 Showing results for http://dbpedia.org/resource/Hristo_Stoichkov [View SPARQL Query](#)

	entity	name	description	birthDate	birthPlace	gender	score
1	dbr:Hristo_Stoichkov	"Hristo Stoichkov"@en	"Bulgarian footballer"@en	"1966-02-08"^^xsd:date	dbr:Plovdiv	"male"@en	"1.0000000005002232"^^xsd:double
2	dbr:Yasen_Petrov	"Yasen Petrov"@en	"Bulgarian footballer"@en	"1968-06-23"^^xsd:date	dbr:Plovdiv	"male"@en	"0.44738577701136845"^^xsd:double
3	dbr:Hristo_Kolev	"Hristo Kolev"@en	"Bulgarian footballer"@en	"1964-09-21"^^xsd:date	dbr:Plovdiv	"male"@en	"0.4371604521068489"^^xsd:double
4	dbr:Petar_Kurdov	"Petar Kurdov"@en	"Bulgarian footballer"@en	"1961-03-13"^^xsd:date	dbr:Plovdiv	"male"@en	"0.3918409885415685"^^xsd:double
5	dbr:Hristo_Koilov	"Hristo Koilov"@en	"Bulgarian footballer"@en	"1969-09-23"^^xsd:date	dbr:Plovdiv	"male"@en	"0.37515620507838054"^^xsd:double

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?”

Search in similarity

Search type: **Analogical** ▾

search options ▾

If `http://nations.com/United_States -> http://nations.com/United_States_dollar` then `http://nations.com/Mexico -> ?` [View SPARQL Query](#)

	resultValue	score
1	http://nations.com/Mexican_peso	"0.26835414867610574"^^xsd:double

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.

5.8.3.7 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-warnspoland-could-hold-brexit-style-eu-referendum> <pubo:formattedDate> 1/11/
↪ 2018
<express:POLAND-s-bid-for-World-War-2-reparations-is-bolstered-by-a-poll-which-found-that-a-majorit>
↪ <pubo:formattedDate> 1/6/2018
```

Naturally we would expect the first news article to be more similar to the second one than to the third one, not only based on their topics - Poland’s relationship with the EU - but also because of their dates. However, the normal Predication index would not take into account the similarity of the dates, and all news would have fairly close scores. In order to handle this type of scenario, we can first create a Text similarity index. It will find that the dates of the three articles are similar, and will then use this information when building the Predication index.

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*, enable *Autocomplete index*, and create an index for *allNews* using the *Build Now* button.

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 can not be used for similarity searching. The index will include all literals returned by the `?documentText` variable from the Data query.

Create text similarity index
Create predication index

Index name

more options ^

Semantic Vectors create index parameters
See the full list of supported parameters

Stop words

Analyzer Class

☒ Literal index

Make sure to filter out the mentions, so the data in the Literal index only contains the news. When creating the index, use the following Data query:

```
SELECT ?documentID ?documentText {
  ?documentID ?p ?documentText .
  filter(isLiteral(?documentText))
  filter (?p != <http://factforge.net/ff2016-mapping/mentionsEntity>)
}
```

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.

Index name

more options ^

Semantic Vectors create index parameters
See the full list of supported parameters

Input Literal Index

Since you don't want to look at mentions, and in this sense the default data format is useless, you need to filter them out from the data used in the predication index. Add the following Data query:

```
SELECT ?subject ?predicate ?object
WHERE {
    ?subject ?predicate ?object .
    filter (?predicate != <http://factforge.net/ff2016-mapping/mentionsEntity>)
    filter (?predicate != <http://ontology.ontotext.com/publishing#creationDate>)
}
```

For the purposes of the test, we want to also display the new formatted date when retrieving data. Go to the search query tab and add the following query:

```
PREFIX :<http://www.ontotext.com/graphdb/similarity/>
PREFIX inst:<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;
        :searchParameters ?parameters;
        ?resultType ?result .
    ?result :value ?entity ;
        :score ?score .
    ?entity pubo:content ?content .
    ?entity pubo:formattedDate ?date .
}
```

With those two queries in place, the data returned from the index should be more useful. Create your hybrid predication index and wait for the process to be completed. Then, open it and run a query for “donald tusk”, selecting the express article about “Polexit” from the Autocomplete suggest box. You will see that the first results are related to the Polexit and dated the same.

Indexing behavior

When building the Literal index, it is 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

5.8.3.8 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.

5.8.4 JavaScript functions

What's in this document?

- *How to register JS function*
- *How to remove JS function*

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 a special namespace called `<http://www.ontotext.com/js#>`

5.8.4.1 How to register JS function

JS functions are initialised 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#>
SELECT ?s ?o {
  ?s jsfn:enum ?o
}

```

	s	o
1	isPalindrome	function isPalindrome(str) { if (!(str instanceof java.lang.String)) return false; var rev = reverse(str); return str.equals(rev); }
2	reverse	function reverse(str) { return str.split("").reverse().join(""); }

<http://www.ontotext.com/js#enum> is a reserved predicate IRI to list the available JS functions.

The following example registers a single function to return yesterday's date:

```

PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
  [] jsfn:register '''
    function getDateYesterday() {
      var date = new Date();
      date.setDate(date.getDate() - 1);
      return date.toJSON().slice(0,10);
    }
  '''
}

```

We can then use this function in a regular SPARQL query, e.g. to retrieve data created the other day:

```

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 = xsd:date(jsfn:getDateYesterday()))
}

```

Note: The projected ?date is filtered by type and dynamically assigned value - xsd:date and the output of the JS function, respectively.

5.8.4.2 How to remove JS function

Deregistering JavaScript function is handled in the same fashion as registering one. What is different is the predicate used in the INSERT statement - <http://www.ontotext.com/js#remove>

```

PREFIX jsfn:<http://www.ontotext.com/js#>
INSERT DATA {
  [] jsfn:remove "getDateYesterday"
}

```

Once removed, the function should be listed as UNDEFINED

	s	o
1	isPalindrome	<pre>function isPalindrome(str) { if (!(str instanceof java.lang.String)) return false; var rev = reverse(str); return str.equals(rev); }</pre>
2	reverse	<pre>function reverse(str) { return str.split("").reverse().join(""); }</pre>
3	getDateYesterday	UNDEFINED

Note: If multiple function definitions have been registered by a single insert, removing one of these functions will remove the rest of the functions added by that insert request.

Several of the plugins are user-defined indexes that you can find [here](#).

5.9 User-defined Indexes

These indexes 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.

5.9.1 Autocomplete index

What's in this document?

- [Autocomplete in the SPARQL editor](#)
- [Autocomplete in the View resource on Home page](#)

The *Autocomplete index* offers suggestions for the URIs local names in the *SPARQL editor* and the *View Resource* page.

It is disabled by default. Go to *Setup -> Autocomplete* to enable it. GraphDB indexes all URIs in the repository by splitting their local names into words, for example, `subPropertyOf` is split into `sub+Property+Of`. This way, when you search for a word, the Autocomplete finds URIs with local names containing the symbols that you typed in the editor.

Autocomplete index ⓘ

Click to enable autocomplete

Autocomplete is **OFF**

If you get strange results and you think the index is broken, use the *Build Now* button.

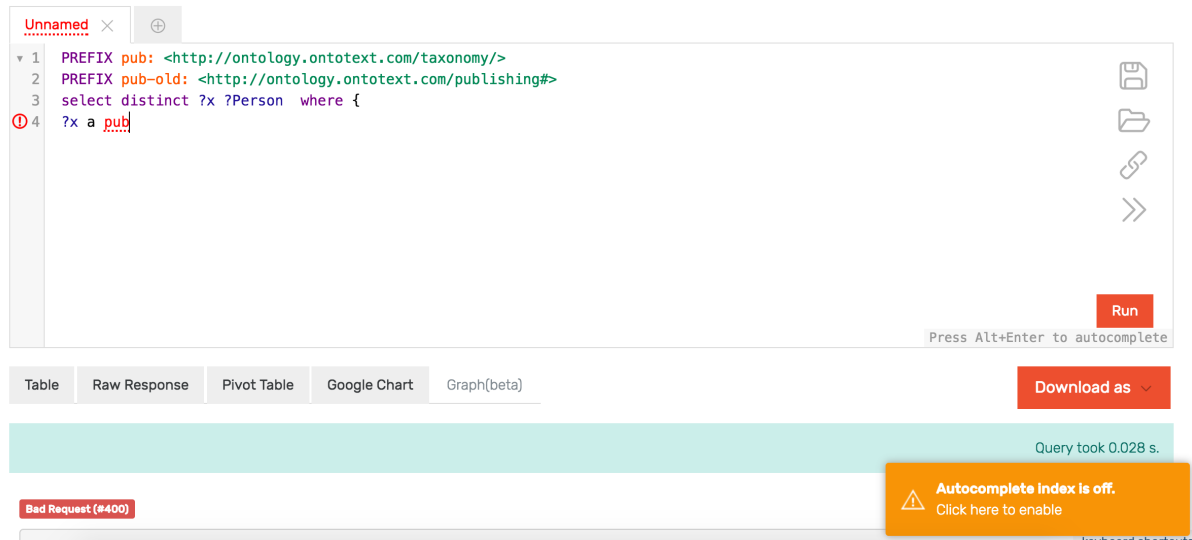
Autocomplete index ⓘ

Autocomplete is **ON**

Index status is **Ready**

Build Now

If you try to use autocompletion before it is enabled, a tooltip warns you that the autocomplete index is off and provides a link for building the index.

SPARQL Query & Update Editor only Editor and results Results only 


```

1 PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
2 PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
3 select distinct ?x ?Person where {
4 ?x a pub

```

Bad Request (#400)

Autocomplete index is off. Click here to enable

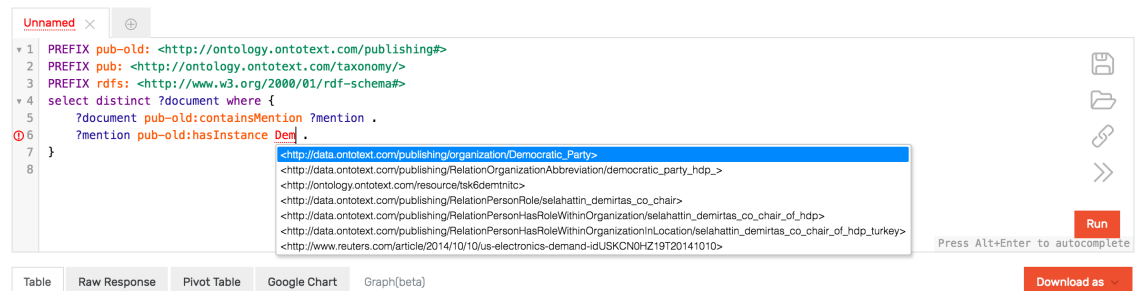
Query took 0.028 s.

Download as

5.9.1.1 Autocomplete in the SPARQL editor

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 in all URIs

SPARQL Query & Update Editor only Editor and results Results only 


```

1 PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
2 PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 select distinct ?document where {
5 ?document pub-old:containsMention ?mention .
6 ?mention pub-old:hasInstance Dem
7 }
8

```

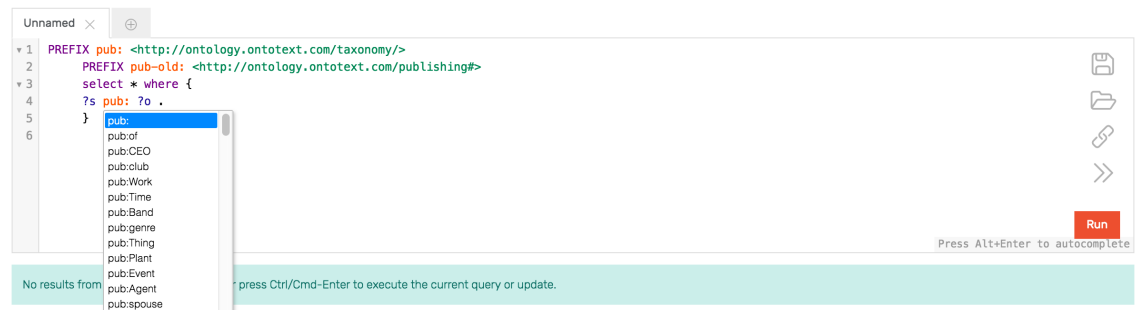
Autocomplete suggestions for ?mention:

- <http://data.ontotext.com/publishing/organization/Democratic_Party>
- <http://data.ontotext.com/publishing/RelationOrganizationAbbreviation/democratic_party_hdp_>
- <http://ontology.ontotext.com/resource/tsk6demnitc>
- <http://data.ontotext.com/publishing/RelationPersonRole/selehattin_demirtas_co_chair>
- <http://data.ontotext.com/publishing/RelationPersonHasRoleWithinOrganization/selehattin_demirtas_co_chair_of_hdp>
- <http://data.ontotext.com/publishing/RelationPersonHasRoleWithinOrganizationInLocation/selehattin_demirtas_co_chair_of_hdp_turkey>
- <http://www.reuters.com/article/2014/10/10/us-electronics-demand-idUSKCN0hZ19T20141010>

Run

Download as

- search only for URIs that start with a certain prefix

SPARQL Query & Update Editor only Editor and results Results only 


```

1 PREFIX pub: <http://ontology.ontotext.com/taxonomy/>
2 PREFIX pub-old: <http://ontology.ontotext.com/publishing#>
3 select * where {
4 ?s pub: ?o .
5 }
6

```

Autocomplete suggestions for ?o:

- pub:of
- pub:CEO
- pub:club
- pub:Work
- pub:Time
- pub:Band
- pub:genre
- pub:Thing
- pub:Plant
- pub:Event
- pub:Agent
- pub:spouse

No results from

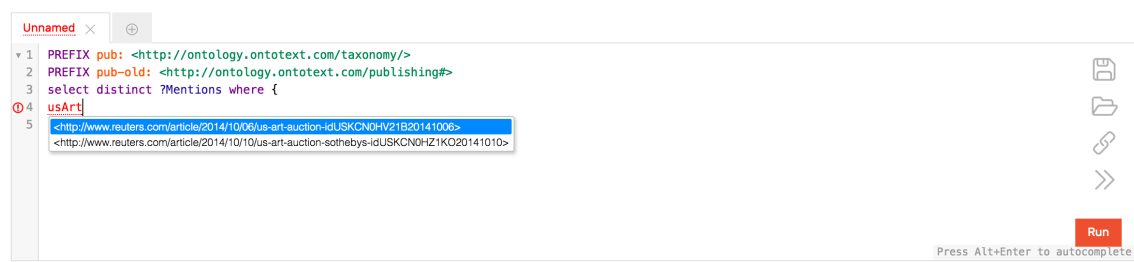
press Ctrl/Cmd-Enter to execute the current query or update.

Run

- search for more than one word

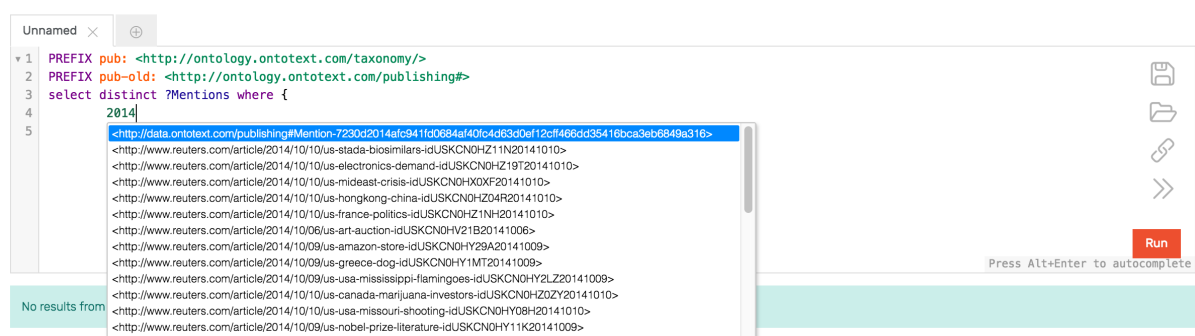
Tip: Just start writing the words one after another without spaces, e.g., “pngOnto”, and the index smartly splits them.

SPARQL Query & Update



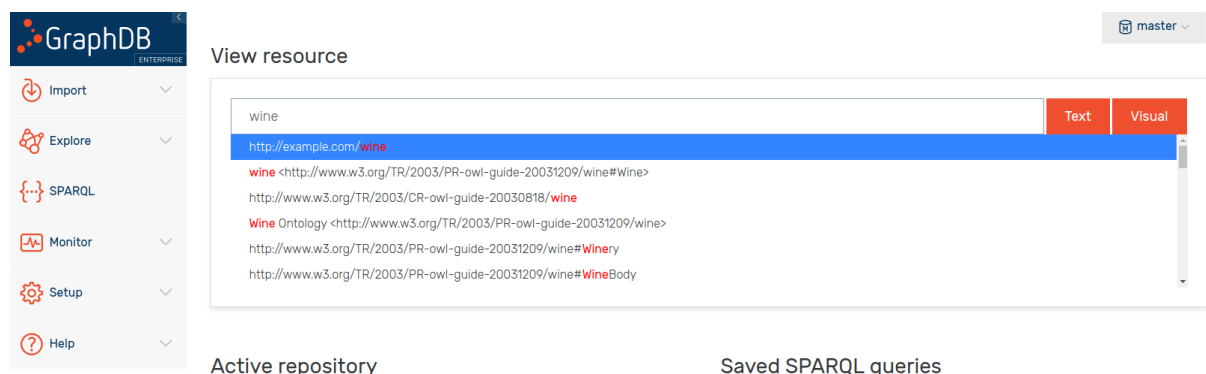
- search for numbers

SPARQL Query & Update



5.9.1.2 Autocomplete in the View resource on Home page

To use the autocompletion feature to find a resource, go to the GraphDB home page, and start typing in the *Explore* -> *View resource* field. The retrieved results can be in text format or in the form of a visual graph, with the latter automatically taking you to the *Visual graph* view.



5.9.2 GeoSPARQL support

What's in this document?

- What is GeoSPARQL
- Usage
 - Plugin control predicates
 - * Enable plugin
 - * Disable plugin

- * *Check the current configuration*
- * *Update the current configuration*
- * *Speed up the building and rebuilding of the GeoSPARQL index*
- * *Force reindex geometry data*
- * *Ignore errors on indexing*
- *GeoSPARQL extensions*
- *GeoSPARQL examples*
 - *Example 1*
 - * *Using a function*
 - * *Using a predicate*
 - * *Example 1 result*
 - *Example 2*
 - * *Using a function*
 - * *Using a predicate*
 - * *Example 2 result*
 - *Example 3*
 - * *Using a function*
 - * *Using a predicate*
 - * *Example 3 result*
 - *Example 4*
 - * *Example 4 result*
 - *Example 5*
 - * *Example 5 result*

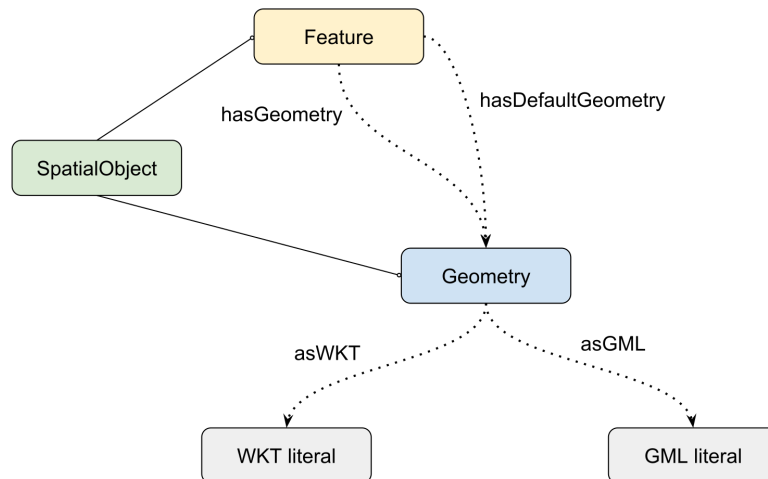
5.9.2.1 What is GeoSPARQL

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:



5.9.2.2 Usage

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 reindexes any updates.

```

PREFIX : <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
  _:s :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.

```

PREFIX : <http://www.ontotext.com/plugins/geosparql#>
  
```

```
INSERT DATA {  
  _:s :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:

```
PREFIX : <http://www.ontotext.com/plugins/geosparql#>  
  
SELECT DISTINCT * WHERE {  
  _:s :currentPrefixTree ?tree;  
      :currentPrecision ?precision;  
      :ramBufferSizeMB ?ramBufferSizeMB;  
      :maxBufferedDocs ?maxBufferedDocs.  
}
```

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 $\pm 2.5\text{km}$ on the equator. When increased to 20 the precision goes down to $\pm 6\text{m}$ accuracy. Respectively, the geohash prefix tree with precision 11 results $\pm 1\text{m}$.

```
PREFIX : <http://www.ontotext.com/plugins/geosparql#>  
  
INSERT DATA {  
  _:s :prefixTree "quad"; #geohash  
      :precision "25";  
      :ramBufferSizeMB "64";  
      :maxBufferedDocs "2000".  
}
```

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, maximum 512.
- `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.

```
PREFIX : <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
  _:s :forceReindex ""
}
```

Ignore errors on indexing

```
PREFIX : <http://www.ontotext.com/plugins/geosparql#>

INSERT DATA {
  _:s :ignoreErrors "true"
}
```

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 offers a few useful extensions. Those are based on the USeekMSail. All extension functions have the ext prefix, where ext stands for <http://rdf.useekm.com/ext#>

Function	Description
double ext:area(geometry)	Calculates the area of the surface of the geometry.
point ext:closestPoint(geometry, geometry)	For two given geometries, computes the point on the first geometry that is closest to the second geometry.
bool ext:containsProperly(geometry, geometry)	Tests if the first geometry properly contains the second geometry. <i>Geom1 containsProperly geom2</i> if all geom1 contains geom2 and the boundaries of the two geometries <i>do not</i> intersect.
bool ext:coveredBy(geometry, geometry)	Tests if the first geometry is covered by the second geometry. <i>Geom1 is covered by geom2</i> if every point of geom1 is a point of geom2.
bool ext:covers(geometry, geometry)	Tests if the first geometry covers the second geometry. <i>Geom1 covers geom2</i> if every point of geom2 is a point of geom1.
double ext:hausdorffDistance(geometry, geometry)	Measures the degree of similarity between two geometries. The measure is <i>normalized</i> to lie in the range [0, 1]. Higher measures indicate a greater degree of similarity.
line ext:shortestLine(geometry, geometry)	Computes the shortest line between two geometries. Returns it as a LineString object.
geometry ext:simplify(geometry, double)	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peucker algorithm .
geometry ext:simplifyPreserveTopology(geometry, double)	Given a maximum deviation from the curve, computes a simplified version of the given geometry using the Douglas-Peucker algorithm . Will avoid creating derived geometries (polygons in particular) that are invalid.
bool ext:isValid(geometry)	Checks whether the input geometry is a valid geometry.

5.9.2.3 GeoSPARQL examples

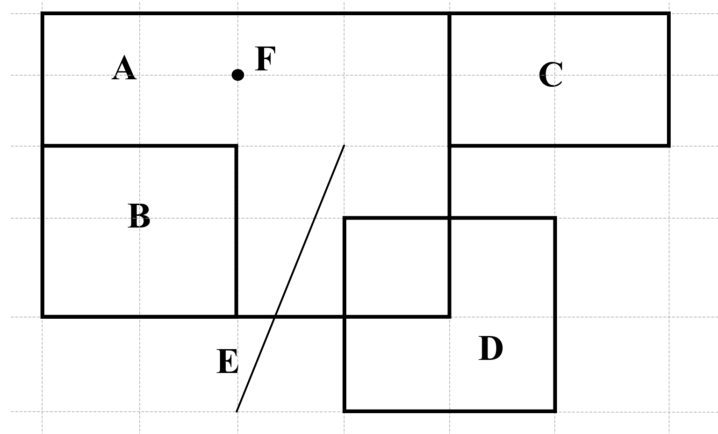
This section contains examples of SELECT queries on geographic data.

Examples 1, 2 and 3 have a variant using a function (corresponding to the same example in the GeoSPARQL specification), as well as a variant where the function is substituted with a predicate. Examples 4 and 5 use a predicate and correspond to the same examples in the specification.

To run the examples you need to:

- Download and import the file `geosparql-example.rdf`.
- Enable the GeoSPARQL plugin.

The data defines the following spatial objects:



Example 1

Find all features that feature my:A contains, where spatial calculations are based on my:hasExactGeometry.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  my:A my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfContains(?aWKT, ?fWKT) && !sameTerm(?aGeom, ?fGeom))
}
```

Using a predicate

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  my:A my:hasExactGeometry ?aGeom .
  ?f my:hasExactGeometry ?fGeom .
  ?aGeom geo:sfContains ?fGeom .
  FILTER (!sameTerm(?aGeom, ?fGeom))
}
```

Example 1 result

?f
my:B
my:F

Example 2

Find all features that are within a transient bounding box geometry, where spatial calculations are based on `my:hasPointGeometry`.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  ?f my:hasPointGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (geof:sfWithin(?fWKT, '''
    <http://www.opengis.net/def/crs/OGC/1.3/CRS84>
    Polygon ((-83.4 34.0, -83.1 34.0,
              -83.1 34.2, -83.4 34.2,
              -83.4 34.0))
    '''^geo:wktLiteral))
}
```

Using a predicate

Note: Using geometry literals in the object position is a GraphDB extension and not part of the GeoSPARQL specification.

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  ?f my:hasPointGeometry ?fGeom .
  ?fGeom geo:sfWithin '''
    <http://www.opengis.net/def/crs/OGC/1.3/CRS84>
    Polygon ((-83.4 34.0, -83.1 34.0,
              -83.1 34.2, -83.4 34.2,
              -83.4 34.0))
    '''^geo:wktLiteral
}
```

Example 2 result

?f
my:D

Example 3

Find all features that touch the union of feature `my:A` and feature `my:D`, where computations are based on `my:hasExactGeometry`.

Using a function

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  my:A my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  my:D my:hasExactGeometry ?dGeom .
  ?dGeom geo:asWKT ?dWKT .
  FILTER (geof:sfTouches(?fWKT, geof:union(?aWKT, ?dWKT)))
}
```

Using a predicate

```
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  my:A my:hasExactGeometry ?aGeom .
  ?aGeom geo:asWKT ?aWKT .
  my:D my:hasExactGeometry ?dGeom .
  ?dGeom geo:asWKT ?dWKT .
  BIND(geof:union(?aWKT, ?dWKT) AS ?union) .
  ?fGeom geo:sfTouches ?union
}
```

Example 3 result

?f
my:C

Example 4

Find the 3 closest features to feature my:C, where computations are based on my:hasExactGeometry.

```
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
PREFIX my: <http://example.org/ApplicationSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?f
WHERE {
  my:C my:hasExactGeometry ?cGeom .
  ?cGeom geo:asWKT ?cWKT .
  ?f my:hasExactGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  FILTER (?fGeom != ?cGeom)
```

```
}  
ORDER BY ASC(geof:distance(?cWKT, ?fWKT, uom:metre))  
LIMIT 3
```

Example 4 result

?f
my:A
my:E
my:D

Note: The example in the GeoSPARQL specification has a different order in the result: my:A, my:D, my:E. In fact, feature my:E is closer than feature my:D even if that does not seem obvious from the drawing of the objects. my:E's closest point is 0.1° to the West of my:C, while my:D's closest point is 0.1° to the South. At that latitude and longitude the difference in terms of distance is larger in latitude, hence my:E is closer.

Example 5

Find all features or geometries that overlap feature my:A.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX my: <http://example.org/ApplicationSchema#>  
  
SELECT ?f  
WHERE {  
    ?f geo:sf0overlaps my:AExactGeom  
}
```

Example 5 result

?f
my:D
my:DExactGeom

Note: The example in the GeoSPARQL specification has additional results my:E and my:EExactGeom. In fact, my:E and my:EExactGeom do not overlap my:AExactGeom because they are of different dimensions (my:AExactGeom is a Polygon and my:EExactGeom is a LineString) and the *overlaps* relation is defined only for objects of the same dimension.

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.

5.9.3 Geo-spatial extensions

What's in this document?

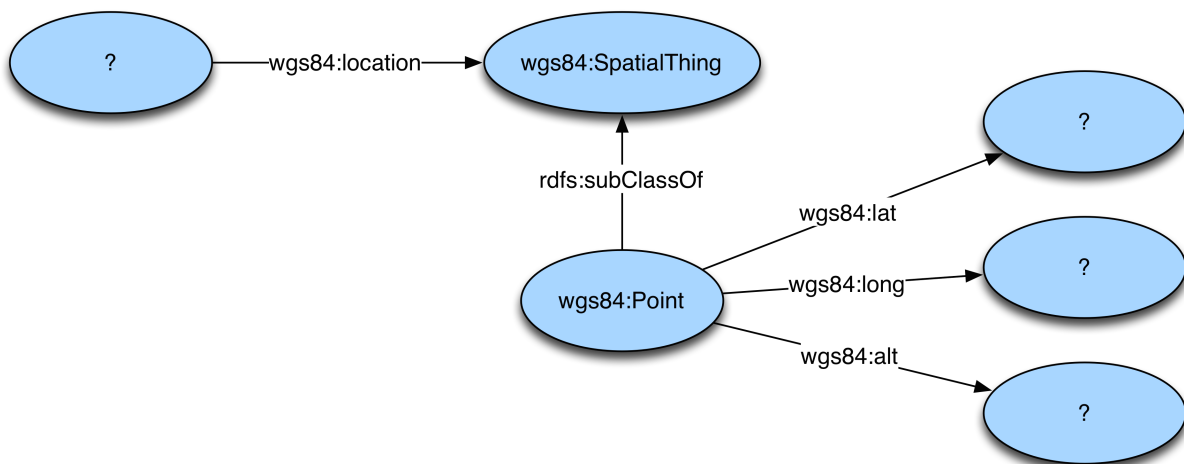
- *What are geo-spatial extensions*
- *How to create a geo-spatial index*

- *Geo-spatial query syntax*
- *Extension query functions*
- *Implementation details*

5.9.3.1 What are geo-spatial extensions

GraphDB provides support for 2-dimensional geo-spatial data that uses the [WGS84 Geo Positioning RDF vocabulary](#) (World Geodetic System 1984). Specialised 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:

Element	Description
SpatialThing	A class for representing anything with a spatial extent, i.e., size, shape or position.
Point	A class for representing a point (relative to Earth) defined by latitude, longitude (and altitude). <code>subClassOf http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</code>
location	The relation between a thing and where it is. Range <code>SpatialThing</code> <code>subPropertyOf http://xmlns.com/foaf/0.1/based_near</code>
lat	The WGS84 latitude of a <code>SpatialThing</code> (decimal degrees). domain <code>http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</code>
long	The WGS84 longitude of a <code>SpatialThing</code> (decimal degrees). domain <code>http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</code>
lat_long	A comma-separated representation of a latitude, longitude coordinate.
alt	The WGS84 altitude of a <code>SpatialThing</code> (decimal meters above the local reference ellipsoid). domain <code>http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing</code>

5.9.3.2 How to create a geo-spatial index

Execute the following INSERT query:

```
PREFIX ontogeo: <http://www.ontotext.com/owlim/geo#>
INSERT DATA { _:b1 ontogeo:createIndex _:b2. }
```

If all geo-spatial 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 geo-spatial data in the repository, i.e., no statements describing resources with latitude and longitude properties, this update query will fail.

5.9.3.3 Geo-spatial query syntax

The Geo-spatial 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 geo-spatial predicates, supported by GraphDB are treated differently by the query engine.

The following special syntax is supported when evaluating SPARQL queries. All descriptions use the namespace: `omgeo: <http://www.ontotext.com/owlim/geo#>`

Construct	Nearby (lat long distance)
Syntax	<code>?point omgeo:nearby(?lat ?long ?distance)</code>
Description	This statement pattern will evaluate to true, if the following constraints hold: • <code>?point geo:lat ?plat .</code> • <code>?point geo:long ?plong .</code> • Shortest great circle distance from <code>(?plat, ?plong)</code> to <code>(?lat, ?long)</code> <code><= ?distance</code> Such a construction uses the geo-spatial indexes to find bindings for <code>?point</code>, which lie within the defined circle. Constants are allowed for any of <code>?lat ?long ?distance</code>, where latitude and longitude are specified in decimal degrees and distance is specified in either kilometers ('km' suffix) or miles ('mi' suffix). If the units are not specified, then 'km' is assumed.
Restrictions	Latitude is limited to the range -90 (South) to 90 (North). Longitude is limited to the range -180 (West) to +180 (East).
Examples	Find the names of airports within 50 miles of Seoul: <pre>PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT distinct ?airport WHERE { ?base geo-ont:name "Seoul" . ?base geo-pos:lat ?latBase . ?base geo-pos:long ?longBase . ?link omgeo:nearby(?latBase ?longBase "50mi") . ?link geo-ont:name ?airport . ?link geo-ont:featureCode geo-ont:S.AIRP . }</pre>

Construct	Within (rectangle)
Syntax	<code>?point omgeo:within(?lat1 ?long1 ?lat2 ?long2)</code>
Description	This statement pattern is used to test/find points that lie within the rectangle specified by diagonally opposite corners <code>?lat1 ?long1</code> and <code>?lat2 ?long2</code>. The corners of the rectangle must be either constants or bound values. It will evaluate to true, if the following constraints hold: • <code>?point geo:lat ?plat .</code> • <code>?point geo:long ?plong .</code> • <code>?lat1 <= ?plat <= ?lat2</code> • <code>?long1 <= ?plong <= ?long2</code> 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 <code>?lat1 ?long1 ?lat2 ?long2</code>, where latitude and longitude are specified in decimal degrees. If <code>?point</code> 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: <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT ?feature ?lat ?long WHERE { ?link omgeo:within(45.85 9.15 48.61 13.18) . ?link geo-ont:featureCode geo-ont:R.TNL . ?link geo-ont:name ?feature . ?link geo-pos:lat ?lat . ?link geo-pos:long ?long . }</pre>

Construct	Within (polygon)
Syntax	<code>?point omgeo:within(?lat1 ?long1 ... ?latN ?longN)</code>
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: • <code>?point geo:lat ?plat</code> . • <code>?point geo:long ?plong</code> . • the position <code>?plat ?plong</code> 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 <code>?point</code> 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: <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT ?feature ?lat ?long WHERE { ?link omgeo:within("51.45" "-2.59" "54.99" "-3.06" "55.81" "-2.03" "52.74" "1.68" "51.17" "1.41") . ?link geo-ont:featureCode geo-ont:S.CAVE . ?link geo-ont:name ?feature . ?link geo-pos:lat ?lat . ?link geo-pos:long ?long . }</pre>

5.9.3.4 Extension query functions

At present, there is just one SPARQL extension function:

Function	Distance function
Syntax	<code>double omgeo:distance(?lat1, ?long1, ?lat2, ?long2)</code>
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: <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT distinct ?airport_name WHERE { ?a1 geo-ont:name "Bournemouth" . ?a1 geo-pos:lat ?lat1 . ?a1 geo-pos:long ?long1 . ?airport omgeo:nearby(?lat1 ?long1 "80mi") . ?airport geo-ont:name ?airport_name . ?airport geo-ont:featureCode geo-ont:S.AIRP . ?airport geo-pos:lat ?lat2 . ?airport geo-pos:long ?long2 . ?a2 geo-ont:name "Brize Norton" . ?a2 geo-pos:lat ?lat3 . ?a2 geo-pos:long ?long3 . FILTER(omgeo:distance(?lat2, ?long2, ?lat3, ?long3) < 80) } ORDER BY ASC(omgeo:distance(?lat2, ?long2, ?lat3, ?long3)) </pre>

5.9.3.5 Implementation details

Knowing the implementation's algorithms and assumptions allow you to make the best use of the GraphDB geo-spatial extensions.

The following aspects are significant and can affect the expected behaviour 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 $-90 \leq \text{latitude} \leq +90$ and $-180 \leq \text{longitude} \leq +180$;
- Distances must be in units of kilometers (suffix 'km') or statute miles (suffix 'mi'). If the suffix is omitted, kilometers are assumed;
- `omgeo:within( rectangle )` construct uses a 'rectangle' whose edges are lines of latitude and longitude, so the north-south distance is constant, and the rectangle described forms a band around the Earth, which starts and stops at the given longitudes;
- `omgeo:within( polygon )` joins vertices with straight lines on a cylindrical projection of the Earth tangential to the equator. A straight line starting at the point under test and continuing East out of the polygon is examined to see how many polygon edges it intersects. If the number of intersections is even, then the point is outside the polygon. If the number of intersections is odd, the point is inside the polygon. With the current algorithm, the order of vertices is not relevant (clockwise or anticlockwise);
- `omgeo:within()` may not work correctly when the region (polygon or rectangle) spans the +/-180 meridian;
- `omgeo:nearby()` uses the great circle distance between points.

5.9.4 Lucene Full-text Search

What's in this document?

- *RDF search*
 - *Usage*
 - *Parameters*
 - *Creating an index*
 - *Incremental update*

Hint: [Apache Lucene](#) is a high-performance, full-featured text search engine written entirely in Java. GraphDB supports FTS capabilities using Lucene with a variety of indexing options and the ability to simultaneously use multiple, differently configured indices in the same query.

Full-text search (FTS) concerns retrieving text documents out of a large collection by keywords or, more generally, by tokens (represented as sequences of characters). Formally, the query represents an unordered set of tokens and the result is a set of documents, relevant to the query. In a simple FTS implementation, relevance is Boolean: a document is either relevant to the query, if it contains all the query tokens, or not. More advanced FTS implementations deal with a degree of relevance of the document to the query, usually judged on some sort of measure of the frequency of appearance of each of the tokens in the document, normalised, versus the frequency of their appearance in the entire document collection. Such implementations return an ordered list of documents, where the most relevant documents come first.

FTS and structured queries, like these in database management systems (DBMS), are different information access methods based on a different query syntax and semantics, where the results are also displayed in a different form. FTS systems and databases usually require different types of indices, too. The ability to combine these two types of information access methods is very useful for a wide range of applications. Many relational DBMS support some sort of FTS (which is integrated in the SQL syntax) and maintain additional indices that allow efficient evaluation of FTS constraints.

Typically, a relational DBMS allows you to define a query, which requires specific tokens to appear in a specific column of a specific table. In SPARQL, there is no standard way for the specification of FTS constraints. In general, there is neither a well-defined nor commonly accepted concept for FTS in RDF data. Nevertheless, some semantic repository vendors offer some sort of FTS in their engines.

5.9.4.1 RDF search

The GraphDB FTS implementation, called ‘RDF Search’, is based on Lucene. It enables GraphDB to perform complex queries against character data, which significantly speeds up the query process. RDF Search allows for efficient extraction of RDF resources from huge datasets, where ordering of the results by relevance is crucial.

Its main features are:

- FTS query form - List of tokens (with Lucene query extensions);
- Result form - Ordered list of URIs;
- Textual Representation - Concatenation of text representations of nodes from the so called ‘molecule’ (1-step neighbourhood in a graph) of the URI;
- Relevance - Vector-space model, reflecting the degree of relevance of the text and the RDF rank of the URI;
- Implementation - The Lucene engine is integrated and used for indexing and search.

Usage

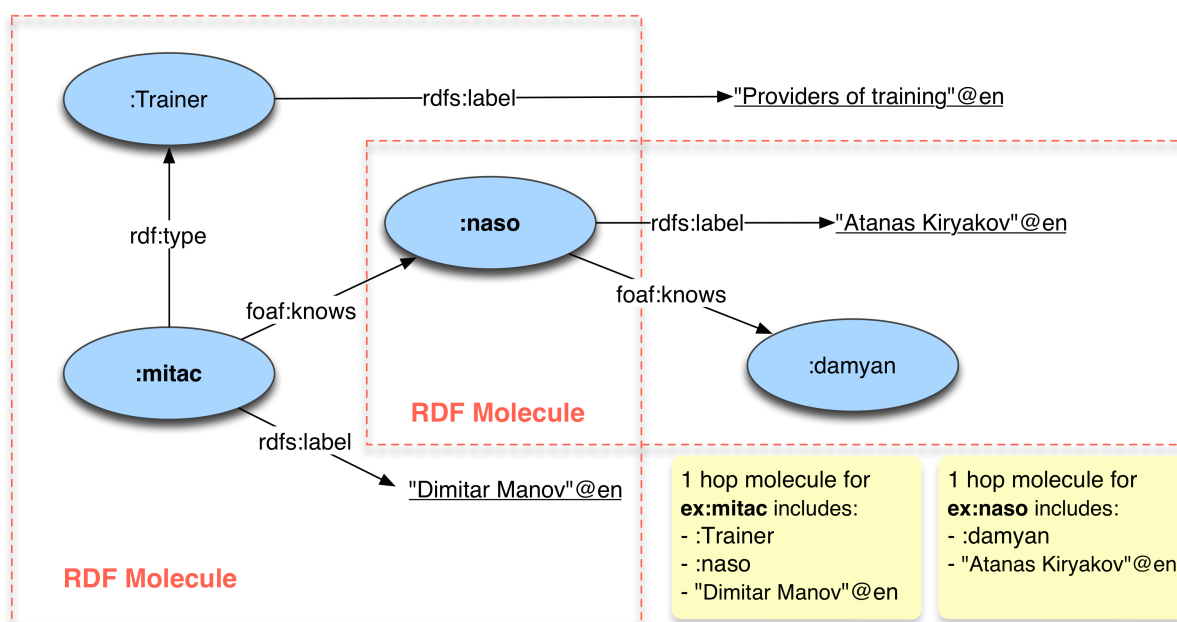
In order to use the FTS in GraphDB, first a Lucene index must be computed. Before it is created, each index can be parametrised in a number of ways, using SPARQL ‘control’ updates.

This provides the ability to:

- select what kinds of nodes are indexed (URIs/literals/blank-nodes);
- select what is included in the ‘molecule’ associated with each node;
- select literals with certain language tags;
- choose the size of the RDF ‘molecule’ to index;
- choose whether to boost the relevance of the nodes using RDF Rank values;
- select alternative analysers;
- select alternative scorers.

In order to use the indexing behaviour of Lucene, a text document must be created for each node in the RDF graph to be indexed. This text document is called an ‘RDF molecule’ and is made up of other nodes reachable via the predicates that connect the nodes to each other. Once a molecule has been created for each node, Lucene generates an index over these molecules. During search (query answering), Lucene identifies the matching molecules and GraphDB uses the associated nodes as variables substitutions, when evaluating the enclosing SPARQL query.

The scope of an RDF molecule includes the starting node and its neighbouring nodes, which are reachable via the specified number of predicate arcs. For each Lucene index, it can be specified what type of nodes are indexed and what type of nodes are included in the molecule. Furthermore, the size of the molecule can be controlled by specifying the number of allowed traversals of predicate arcs starting from the molecule centre (the node being indexed).



Note: Blank nodes are never included in the molecule. If a blank node is encountered, the search is extended via any predicate to the next nearest entity and so on. Therefore, even when the molecule size is 1, entities reachable via several intermediate predicates can still be included in the molecule if all the intermediate entities are blank nodes.

Parameters

Exclude

Predicate: <http://www.ontotext.com/owlim/lucene#exclude>

Default: <none>

Description: Provides a regular expression to identify nodes, which will be excluded from the molecule.

Note that for literals and URI local names the regular expression is case-sensitive.

The example given below will cause matching URIs (e.g., <<http://example.com/uri#helloWorld>>) and literals (e.g., "hello world!") not to be included.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:exclude luc:setParam "hello.*"
}
```

Exclude entities

Predicate: <http://www.ontotext.com/owlim/lucene#excludeEntities>

Default: <none>

Description: A comma/semi-colon/white-space separated list of entities that will NOT be included in an RDF molecule. The example below includes any URI in a molecule, except the two listed.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:excludeEntities luc:setParam
    "http://www.w3.org/2000/01/rdf-schema#Class http://www.example.com/dummy#E1"
}
```

Exclude Predicates

Predicate: <http://www.ontotext.com/owlim/lucene#excludePredicates>

Default: <none>

Description: A comma/semi-colon/white-space separated list of properties that will NOT be traversed in order to build an RDF molecule. The example below prevents any entities being added to an RDF molecule, if they can only be reached via the two given properties.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:excludePredicates luc:setParam
    "http://www.w3.org/2000/01/rdf-schema#subClassOf http://www.example.com/dummy#p1"
}
```

Include

Predicate: <http://www.ontotext.com/owlim/lucene#include>

Default: "literals"

Description: Indicates what kinds of nodes are to be included in the molecule. The value can be a list of values from: URI, literal, centre (the plural forms are also allowed: URIs, literals, centres). The value of centre causes the node for which the molecule is built to be added to the molecule (provided it is not a blank node). This can be useful, for example, when indexing URI nodes with molecules that contain only literals, but the local part of the URI should also be searchable.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:include luc:setParam "literal uri"
}
```

Include entities

Predicate: <http://www.ontotext.com/owlim/lucene#includeEntities>

Default: <none>

Description: A comma/semi-colon/white-space separated list of entities that can be included in an RDF molecule. Any other entities are ignored. The example below builds molecules that only contain the two entities.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:includeEntities luc:setParam
    "http://www.w3.org/2000/01/rdf-schema#Class http://www.example.com/dummy#E1"
}
```

Include predicates

Predicate: <http://www.ontotext.com/owlim/lucene#includePredicates>

Default: <none>

Description: A comma/semi-colon/white-space separated list of properties that can be traversed in order to build an RDF molecule. The example below allows any entities to be added to an RDF molecule, but only if they can be reached via the two given properties.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:includePredicates luc:setParam
    "http://www.w3.org/2000/01/rdf-schema#subClassOf http://www.example.com/dummy#p1"
}
```

Index

Predicate: <http://www.ontotext.com/owlim/lucene#index>

Default: "literals"

Description: Indicates what kinds of nodes are to be indexed. The value can be a list of values from: URI, literal, bnode (the plural forms are also allowed: URIs, literals, bnodes).

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:index luc:setParam "literals, bnodes"
}
```

Languages

Predicate: <http://www.ontotext.com/owlim/lucene#languages>

Default: "" (which is used to indicate that literals with any language tag are used, including those with no language tag)

Description: A comma separated list of language tags. Only literals with the indicated language tags are included in the index. To include literals that have no language tag, use the special value none.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:languages luc:setParam "en,fr,none"
}
```

Molecule size

Predicate: <http://www.ontotext.com/owlim/lucene#moleculeSize>

Default: 0

Description: Sets the size of the molecule associated with each entity. A value of zero indicates that only the entity itself should be indexed. A value of 1 indicates that the molecule will contain all entities reachable by a single ‘hop’ via any predicate (predicates not included in the molecule). Note that blank nodes are never included in the molecule. If a blank node is encountered, the search is extended via any predicate to the next nearest entity and so on. Therefore, even when the molecule size is 1, entities reachable via several intermediate predicates can still be included in the molecule, if all the intermediate entities are blank nodes. Molecule sizes of 2 and more are allowed, but with large datasets it can take a very long time to create the index.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:moleculeSize luc:setParam "1"
}
```

useRDFRank

Predicate: <http://www.ontotext.com/owlim/lucene#useRDFRank>

Default: "no"

Description: Indicates whether the RDF weights (if they have been already computed) associated with each entity should be used as boosting factors when computing the relevance of a given Lucene query. Allowable values are no, yes and squared. The last value indicates that the square of the RDF Rank value is to be used.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
```

```
luc:useRDFRank luc:setParam "yes"
}
```

analyser

Predicate: <http://www.ontotext.com/owlim/lucene#analyzer>

Default: <none>

Description: Sets an **alternative analyser** for processing text to produce terms to index. By default, this parameter has no value and the default analyser used is: `org.apache.lucene.analysis.standard.StandardAnalyzer`. An alternative analyser must be derived from: `org.apache.lucene.analysis.Analyzer`. To use an alternative analyser, use this parameter to identify the name of a Java factory class that can instantiate it. The factory class must be available on the Java virtual machine's classpath and must implement this interface: `com.ontotext.trree.plugin.lucene.AnalyzerFactory`.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:analyzer luc:setParam "com.ex.MyAnalyserFactory"
}
```

Detailed example: In this example, we create two Java classes (`Analyzer` and `Factory`) and then create a Lucene index, using the custom analyser. This custom analyser filters the accents (diacritics), so a search for “Beyonce” finds labels “Beyoncé”.

```
public class CustomAnalyzerFactory implements com.ontotext.trree.plugin.lucene.AnalyzerFactory {
    @Override
    public Analyzer createAnalyzer() {
        CustomAnalyzer ret = new CustomAnalyzer(Version.LUCENE_36);
        return ret;
    }

    @Override
    public boolean isCaseSensitive() {
        return false;
    }
}
```

```
public class CustomAnalyzer extends StopwordAnalyzerBase {
    public CustomAnalyzer(Version matchVersion){
        super(matchVersion, StandardAnalyzer.STOP_WORDS_SET);
    }

    @Override
    protected TokenStreamComponents createComponents(String fieldName, Reader reader) {
        final Tokenizer source = new StandardTokenizer(matchVersion, reader);
        TokenStream tokenStream = source;
        tokenStream = new StandardFilter(matchVersion, tokenStream);
        tokenStream = new LowerCaseFilter(tokenStream);
        tokenStream = new StopFilter(matchVersion, tokenStream, getStopwordSet());
        tokenStream = new ASCIIFoldingFilter(tokenStream);
        return new TokenStreamComponents(source, tokenStream);
    }
}
```

Create the index:

1. Put the two classes in a .jar file, e.g., “com.example”
2. Put the .jar file in the plugins folder (specified by `-Dregister-external-plugins=...`, which by default is under `<TOMCAT-WEBAPPS>/graphdb-server/WEB-INF/classes/plugins`). There has to be some data in the repository.
3. Create the index.

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:analyzer luc:setParam "com.example.CustomAnalyzerFactory" .
  luc:index luc:setParam "uris".
  luc:moleculeSize luc:setParam "1".
  luc:myIndex luc:createIndex "true".
}
```

scorer

Predicate: `http://www.ontotext.com/owlim/lucene#scorer`

Default: `<none>`

Description: Sets an **alternative scorer** that provides boosting values, which adjust the relevance (and hence the ordering) of results to a Lucene query. By default, this parameter has no value and no additional scoring takes place, however, if the `useRDFRank` parameter is set to `true`, then the RDF Rank scores are used. An alternative scorer must implement this interface: `com.ontotext.tree.plugin.lucene.Scorer`. In order to use an alternative scorer, use this parameter to identify the name of a Java factory class that can instantiate it. The factory class must be available on the Java virtual machine's classpath and must implement this interface: `com.ontotext.tree.plugin.lucene.ScorerFactory`.

Example:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:scorer luc:setParam "com.ex.MxScorerFactory"
}
```

Creating an index

Once you have set the parameters for an index, you create and name the index by committing a SPARQL update of this form, where the index name appears as the subject in the triple pattern:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA { luc:myIndex luc:createIndex "true" . }
```

The index name must have the `http://www.ontotext.com/owlim/lucene#` namespace and the local part can contain only alphanumeric characters and underscores.

Creating an index can take some time, although usually no more than a few minutes when the molecule size is 1 or less. During this process, for each node in the repository, its surrounding molecule is computed. Then, each such molecule is converted into a single string document (by concatenating the textual representation of all the nodes in the molecule) and this document is indexed by Lucene. If the RDF Rank weights are used (or an alternative scorer is specified), then the computed values are stored in the Lucene index as a boosting factor that will later on influence the selection order.

To use a custom Lucene index in a SPARQL query, use the index's name as the predicate in a statement pattern, with the Lucene query as the object using the full [Lucene query](#) vocabulary.

The following query produces bindings for ?s from entities in the repository, where the RDF molecule associated with this entity (for the given index) contains terms that begin with “United”. Furthermore, the bindings are ordered by relevance (with any boosting factor):

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
SELECT ?s
WHERE { ?s luc:myIndex "United*" . }
```

The Lucene score for a bound entity for a particular query can be exposed using a special predicate:

```
http://www.ontotext.com/owlim/lucene#score
```

This can be useful when the Lucene query results are ordered in a manner based on but different from the original Lucene score.

For example, the following query orders the results by a combination of the Lucene score and some ontology defined importance value:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
PREFIX ex: <http://www.example.com/myontology#>
SELECT * {
  ?node luc:myIndex "lucene query string" .
  ?node ex:importance ?importance .
  ?node luc:score ?score .
} ORDER BY ( ?score + ?importance )
```

The luc:score predicate works only on bound variables. There is no problem disambiguating multiple indices because each variable is bound from exactly one Lucene index and hence its score.

The combination of ranking RDF molecules together with FTS provides a powerful mechanism for querying/analysing datasets, even when the schema is not known. This allows for keyword-based search over both literals and URIs with the results ordered by importance/interconnectedness.

You can see an example of such ‘RDF Search’ in [FactForge](#).

Detailed example

The following example configuration shows how to index URIs using literals attached to them by a single, named predicate - in this case rdfs:label.

1. Assume the following starting data:

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex:<http://example.com#>
INSERT DATA {
  ex:astonMT rdfs:label "Aston McTalisker" .
  ex:astonMartin ex:link "Aston Martin" .
  <http://www1.aston.ac.uk/> rdfs:label "Aston University"@EN .
}
```

2. Set up the configuration - index URIs by including, in their RDF molecule, all literals that can be reached via a single statement using the rdfs:label predicate:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:index luc:setParam "uris" .
  luc:include luc:setParam "literals" .
  luc:moleculeSize luc:setParam "1" .
  luc:includePredicates luc:setParam "http://www.w3.org/2000/01/rdf-schema#label" .
}
```

3. Create a new index called `luc:myTestIndex` - note that the index name must be in the `<http://www.ontotext.com/owlim/lucene#>` namespace:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA {
  luc:myTestIndex luc:createIndex "true" .
}
```

4. Use the index in a query - find all URIs indexed using the `luc:myTestIndex` index that match the Lucene query `"ast*"`:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
SELECT * {
  ?id luc:myTestIndex "ast*"
}
```

The results of this query are:

?id
http://example.com#astonMT
http://www1.aston.ac.uk/

showing that `ex:astonMartin` is not returned, because it does not have an `rdfs:label` linking it to the appropriate text.

Incremental update

Each Lucene-based FTS index must be recreated from time to time as the indexed data changes. Due to the complex nature of the structure of RDF molecules, rebuilding an index is a relatively expensive operation. Still, indices can be updated incrementally on a per resource basis as directed by the user.

The following control update:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA { <index-name> luc:addToIndex <resource> . }
```

updates the FTS index for the given resource and the given index.

Note: Each index stores the values of the parameters used to define it, e.g., the value of `luc:includePredicates`, therefore there is no need to set them before requesting an incremental update.

The following control update:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
INSERT DATA { <index-name> luc:updateIndex _:b1 . }
```

causes all resources not currently indexed by `<index-name>` to get indexed. It is a shorthand way of batching together index updates for several (new) resources.

The *GraphDB Connectors* are such indexes as well.

5.10 GraphDB connectors

5.10.1 Lucene GraphDB connector

What's in this document?

- *Overview and features*
- *Usage*
- *Setup and maintenance*
 - *Creating a connector instance*
 - * *Using the workbench*
 - * *Using the create command*
 - *Upgrading an index*
 - *Dropping a connector instance*
 - *Listing available connector instances*
 - * *In the Connectors management view*
 - * *With a SPARQL query*
 - *Instance status check*
- *Working with data*
 - *Adding, updating and deleting data*
 - *Simple queries*
 - *Combining Lucene results with GraphDB data*
 - *Entity match score*
 - *Basic facet queries*
 - *Sorting*
 - *Limit and offset*
 - *Snippet extraction*
 - *Total hits*
- *List of creation parameters*
 - *Special field definitions*
 - * *Copy fields*
 - * *Multiple property chains per field*
 - * *Indexing language tags*
 - * *Indexing the URI of an entity*
- *Datatype mapping*
- *Advanced filtering and fine tuning*
 - *Basic entity filter example*
 - *Advanced entity filter example*
- *Overview of connector predicates*
- *Caveats*
 - *Order of control*
- *Upgrading from previous versions*
 - *Migrating from GraphDB 6.2 to 6.6*

- *Migrating from a pre-6.2 version*
- *Changes in field configuration and synchronization*

5.10.1.1 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 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 URI) 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 Lucene 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.

5.10.1.2 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 URI 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 URIs. 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, which 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 : <http://www.ontotext.com/example/wine#> .

:RedWine rdfs:subClassOf :Wine .
:WhiteWine rdfs:subClassOf :Wine .
:RoseWine rdfs:subClassOf :Wine .

:Merlo
  rdf:type :Grape ;
  rdfs:label "Merlo" .

:CabernetSauvignon
  rdf:type :Grape ;
  rdfs:label "Cabernet Sauvignon" .

:CabernetFranc
  rdf:type :Grape ;
  rdfs:label "Cabernet Franc" .

:PinotNoir
  rdf:type :Grape ;
  rdfs:label "Pinot Noir" .

:Chardonnay
  rdf:type :Grape ;
  rdfs:label "Chardonnay" .

:Yoyowine
  rdf:type :RedWine ;
  :madeFromGrape :CabernetSauvignon ;
  :hasSugar "dry" ;
  :hasYear "2013"^^xsd:integer .

:Franvino
  rdf:type :RedWine ;
  :madeFromGrape :Merlo ;
  :madeFromGrape :CabernetFranc ;
  :hasSugar "dry" ;
  :hasYear "2012"^^xsd:integer .

:Noirette
  rdf:type :RedWine ;
  :madeFromGrape :PinotNoir ;
  :hasSugar "medium" ;
  :hasYear "2012"^^xsd:integer .

:Blanquito
```

```
    rdf:type :WhiteWine ;
    :madeFromGrape :Chardonnay ;
    :hasSugar "dry" ;
    :hasYear "2012"^^xsd:integer .

:Rozova
    rdf:type :RoseWine ;
    :madeFromGrape :PinotNoir ;
    :hasSugar "medium" ;
    :hasYear "2013"^^xsd:integer .
```

5.10.1.3 Setup and maintenance

Third-party component versions This version of the Lucene GraphDB Connector uses Lucene version 8.2.0.

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`);
- 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 the *New Connector* button 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



Name*	my_index	
Fields*	Field name* grape Property chain* http://www.ontotext.com/example/wine#madeFromGrape http://www.w3.org/2000/01/rdf-schema#label Default value default value Datatype ☒ Indexed ☒ Stored ☒ Analyzed ☒ Multivalued ☒ Facet Field name* sugar Property chain* http://www.ontotext.com/example/wine#hasSugar Default value default value Datatype ☒ Indexed ☒ Stored ☒ Analyzed ☐ Multivalued ☒ Facet	

[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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :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"
      ]
    },
    {
      "fieldName": "sugar",
      "propertyChain": [
        "http://www.ontotext.com/example/wine#hasSugar"
      ],
      "analysed": false,
      "multivalued": false
    },
    {
      "fieldName": "year",
      "propertyChain": [
        "http://www.ontotext.com/example/wine#hasYear"
      ],
      "analysed": 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 `analysed: false` as well. See [analysed](#) in *Defining fields* for more information.

Upgrading an index

When upgrading to a newer GraphDB version, it might happen that it contains plugins or connectors 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> "lucene" }"
```


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.

Dropping a connector instance

Dropping (deleting) a connector instance removes all references to its external store from GraphDB, as well as all Lucene files associated with it.

The drop command is triggered by a SPARQL INSERT with the `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 : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :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



Are you sure you want to delete this connector?

☒ Force delete

Cancel

Delete

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.

Connector management ⓘ

Lucene

New Connector

Existing connectors

news

dbp-agents

leak-nodes

Fields

Field name

name

Property chain

http://data.ontotext.com/resource/leak/name

Default value

Datatype

xsd:string

☒ Indexed
 ☒ Stored
 ☒ Analyzed
 ☒ Multivalued
 ☒ Facet

Field name

address_direct

Property chain

http://data.ontotext.com/resource/leak/address

Default value

Datatype

xsd:string

☒ Indexed
 ☒ Stored
 ☒ Analyzed
 ☒ Multivalued
 ☒ Facet

Field name

address_registered

Property chain

http://data.ontotext.com/resource/leak/hasRegisteredAddress

http://data.ontotext.com/resource/leak/address

Default value

Datatype

xsd:string

☒ Indexed
 ☒ Stored
 ☒ Analyzed
 ☒ Multivalued
 ☒ Facet

With a SPARQL query

Listing connector instances returns all previously created instances. It is a SELECT query with the `listConnectors` predicate:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>

SELECT ?cntUri ?cntStr {
  ?cntUri :listConnectors ?cntStr .
}
```

`?cntUri` is bound to the prefixed URI 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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>

SELECT ?cntUri ?cntStatus {
  ?cntUri :connectorStatus ?cntStatus .
}
```

`?cntUri` is bound to the prefixed URI of the connector instance, while `?cntStatus` is bound to a string representation of the status of the connector represented by this URI. The status is key-value based.

5.10.1.4 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 Lucene query as the object of the query predicate:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity {
  ?search a inst:my_index ;
    :query "grape:cabernet" ;
    :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 URIs 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 URI. 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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>
PREFIX wine: <http://www.ontotext.com/example/wine#>

SELECT ?entity ?grape ?year {
  ?search a inst:my_index ;
    :query "grape:cabernet" ;
    :entities ?entity .
  ?entity wine:madeFromGrape ?grape .
  ?entity wine:hasYear ?year
}
```

The result looks like this:

?entity	?grape	?year
:Yoyowine	:CabernetSauvignon	2013
:Franvino	:Merlo	2012
:Franvino	:CabernetFranc	2012

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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity ?score {
  ?search a inst:my_index ;
    :query "grape:cabernet" ;
    :entities ?entity .
  ?entity :score ?score
}
```

The result looks like this but the actual score might be different as it depends on the specific Lucene version:

?entity	?score
:Yoyowine	0.9442660212516785
:Franvino	0.7554128170013428

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 : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <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 inst:my_index ;
    :facetFields "year,sugar" ;
    :facets _:f .
  _:f :facetName ?facetName .
  _:f :facetValue ?facetValue .
  _:f :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:

facetName	facetValue	facetCount
year	2012	3
year	2013	2
sugar	dry	3
sugar	medium	2

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 : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity {
  ?search a inst:my_index ;
    :query "year:2013" ;
    :orderBy "-sugar" ;
    :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. 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 “analysed”: `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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity {
  ?search a inst:my_index ;
  :query "sugar:dry" ;
  :offset "1" ;
  :limit "1" ;
  :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:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?entity ?snippetField ?snippetText {
  ?search a inst:my_index ;
```

```

:query "grape:cabernet" ;
:entities ?entity .
?entity :snippets _:s .
_:s :snippetField ?snippetField ;
    :snippetText ?snippetText .
}

```

the query returns the two wines made from Cabernet Sauvignon or Cabernet Franc grapes as well as the respective matching fields and snippets:

?entity	?snippetField	?snippetText
:Yoyowine	grape	Cabernet Sauvignon
:Franvino	grape	Cabernet Franc

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 : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

SELECT ?totalHits {
  ?r a inst:my_index ;
    :query "year:2012" ;
    :totalHits ?totalHits .
}

```

As there are three wines made in 2012, the value 3 (of type `xdd:long`) binds to `?totalHits`.

5.10.1.5 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 in the GraphDB Workbench without any knowledge of JSON.

analyser (string), optional, specifies Lucene analyzer The Lucene Connector supports custom Analyser implementations. They may be specified via the `analyser` parameter whose value must be a fully qualified name of a class that extends `org.apache.lucene.analysis.Analyzer`. The class requires either a default constructor or a constructor with exactly one parameter of type `org.apache.lucene.util.Version`. For example, these two classes are valid implementations:

```
package com.ontotext.example;

import org.apache.lucene.analysis.Analyzer;

public class FancyAnalyzer extends Analyzer {
    public FancyAnalyzer() {
        ...
    }
    ...
}
```

```
package com.ontotext.example;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.util.Version;

public class SmartAnalyzer extends Analyzer {
    public SmartAnalyzer(Version luceneVersion) {
        ...
    }
    ...
}
```

FancyAnalyzer and SmartAnalyzer can then be used by specifying their fully qualified names, for example:

```
...
    "analyser": "com.ontotext.example.SmartAnalyzer",
...
```

types (list of URI), required, specifies the types of entities to sync The RDF types of entities to sync are specified as a list of URIs. At least one type URI is required.

languages (list of string), 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 object), 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.
- **propertyChain (list of URI), 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 URI 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 URIs where at least one URI 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 URI of an entity* for defining a field whose values are populated with the URI of the indexed entity.

- **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 URI. 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".
- **analysed (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 "analysed". This option has no effect for URIs (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.
- **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.

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:

```
...
"fields": [
  {
    "fieldName": "grape",
    "propertyChain": [
      "http://www.ontotext.com/example/wine#madeFromGrape",
      "http://www.w3.org/2000/01/rdf-schema#label"
    ],
    "analysed": true,
  },
],
```

```
{
  "fieldName": "grapeFacet",
  "propertyChain": [
    "@grape"
  ],
  "analysed": false,
}
]
...
```

The snippet creates an analyzed field “grape” and a non-analyzed field “grapeFacet”, both fields are populated with the same values and “grapeFacet” is defined as a copy field that refers to the field “facet”.

Note: The connector handles copy fields in a more optimal way than specifying a field with exactly the same property chain as another field.

Multiple property chains per field

Sometimes, you have to work with data models that define the same concept (in terms of what you want to index in Lucene) with more than one property chain, 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:

```
...
"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-URI `lang()`. The property preceding `lang()` must lead to a literal value. For example,

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :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 language tag of each literal value of the property `http://www.ontotext.com/example#name` into the field `nameLanguage`.

Indexing the URI of an entity

Sometimes you may need the URI 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-URI `$self`. For example,

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :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 URI of each wine into the field `entityId`.

5.10.1.6 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 (URI or literal) and the datatype of literals. The autodetection uses the following mapping:

RDF value	RDF datatype	Lucene type
URI	n/a	StringField
literal	none	TextField
literal	xsd:boolean	StringField with values “true” and “false”
literal	xsd:double	DoubleField
literal	xsd:float	FloatField
literal	xsd:long	LongField
literal	xsd:int	IntField
literal	xsd:dateTime	DateTools.timeToString(), second precision
literal	xsd:date	DateTools.timeToString(), day precision

The datatype mapping can be affected by the synchronization options too, e.g., a non-analyzed field that has `xsd:long` values is indexed with a `StringField`.

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.

5.10.1.7 Advanced filtering and fine tuning

entityFilter (string) The `entityFilter` parameter is 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 entity filter is similar to a `FILTER()` inside a SPARQL query but not exactly the same. Each configured field can be referred to, in the entity filter, by prefixing it with a `?`, much like referring to a variable in SPARQL. Several operators are supported:

Operator	Meaning	Example
?var in (value1, value2, ...)	Tests if the field var's value is one of the specified values. Values that do not match, are treated as if they were not present in the repository.	?status in ("active", "new")
?var not in (value1, value2, ...)	The negated version of the in-operator.	?status not in ("archived")
bound(?var)	Tests if the field var has a valid value. This can be used to make the field compulsory.	bound(?name)
expr1 or expr2	Logical disjunction of expressions expr1 and expr2.	bound(?name) or bound(?company)
expr1 && expr2	Logical conjunction of expressions expr1 and expr2.	bound(?status) && ?status in ("active", "new")
!expr	Logical negation of expression expr.	!bound(?company)
(expr)	Grouping of expressions	(bound(?name) or bound(?company)) && bound(?address)

Note:

- ?var in (...) filters the values of ?var and leaves only the matching values, i.e., it will modify the actual data that will be synchronized to Lucene
- bound(?var) checks if there is any valid value left after filtering operators such as ?var in (...) have been applied

In addition to the operators, there are some constructions that can be used to write filters based not on the values 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 for accessing 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 URI parameter can be a full URI 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. The typical use case is to sync only explicit values: `graph(?a) not in (<http://www.ontotext.com/implicit>)`. 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: `""`.

Entity filters and default values Entity filters can be combined with default values in order to get more flexible

behaviour.

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.

Basic entity filter example

Given the following RDF data:

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

# the entity below will be synchronized because it has a matching value for city: ?city in ("London")
:alpha
  rdfs:type :gadget ;
  :name "John Synced" ;
  :city "London" .

# the entity below will not be synchronized because it lacks the property completely: bound(?city)
:beta
  rdfs:type :gadget ;
  :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
:gamma
  rdfs:type :gadget ;
  :name "Mary Syncless" ;
  :city "Liverpool" .
```

If you create a connector instance such as:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :createConnector '''
  {
    "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"]
      }
    ],
    "entityFilter": "bound(?city) && ?city in (\\\"London\\\")"
  }
  ''' .
}
```

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 : <http://www.ontotext.com/example2#> .

:Berlin
  rdf:type :Location ;
  rdfs:label "Berlin" .

:Mozart
  rdf:type :Person ;
  rdfs:label "Wolfgang Amadeus Mozart" .

:Einstein
  rdf:type :Person ;
  rdfs:label "Albert Einstein" .

:Cannes-FF
  rdf:type :Event ;
  rdfs:label "Cannes Film Festival" .

:Article1
  rdf:type :Article ;
  rdfs:comment "An article about a film about Einstein's life while he was a professor in Berlin." ;
  :taggedWith :Berlin ;
  :taggedWith :Einstein ;
  :taggedWith :Cannes-FF .

:Article2
  rdf:type :Article ;
  rdfs:comment "An article about Berlin." ;
  :taggedWith :Berlin .

:Article3
  rdf:type :Article ;
  rdfs:comment "An article about Mozart's life." ;
  :taggedWith :Mozart .

:Article4
  rdf:type :Article ;
  rdfs:comment "An article about classical music in Berlin." ;
  :taggedWith :Berlin ;
  :taggedWith :Mozart .

:Article5
  rdf:type :Article ;
  rdfs:comment "A boring article that has no tags." .

```

```
:Article6
  rdf:type :Article ;
  rdfs:comment "An article about the Cannes Film Festival in 2013." ;
  :taggedWith :Cannes-FF .
```

Now, if you map this data to Lucene so that the property `:taggedWith x` 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 : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

INSERT DATA {
  inst:my_index :createConnector '''
  {
    "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"]
      },
      {
        "fieldName": "taggedWithLocation",
        "propertyChain": ["http://www.ontotext.com/example2#taggedWith"]
      }
    ],
    "entityFilter": "?taggedWithPerson type in (<http://www.ontotext.com/example2#Person>)
                    && ?taggedWithLocation 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:

Article URI	Value taggedWith- Person	Value taggedWithLo- cation	Explanation
:Article1	:Einstein	:Berlin	:taggedWith has the values :Einstein, :Berlin and :Cannes-FF. The filter leaves only the correct values in the respective fields. The value :Cannes-FF is ignored as it does not match the filter.
:Article2		:Berlin	:taggedWith has the value :Berlin. After the filter is applied, only taggedWithLocation is populated.
:Article3	:Mozart		:taggedWith has the value :Mozart. After the filter is applied, only taggedWithPerson is populated
:Article4	:Mozart	:Berlin	:taggedWith has the values :Berlin and :Mozart. The filter leaves only the correct values in the respective fields.
:Article5			:taggedWith has no values. The filter is not relevant.
:Article6			:taggedWith has the value :Cannes-FF. The filter removes it as it does not match.

This can be checked by issuing a faceted search for taggedWithLocation and taggedWithPerson:

```
PREFIX : <http://www.ontotext.com/connectors/lucene#>
PREFIX inst: <http://www.ontotext.com/connectors/lucene/instance#>

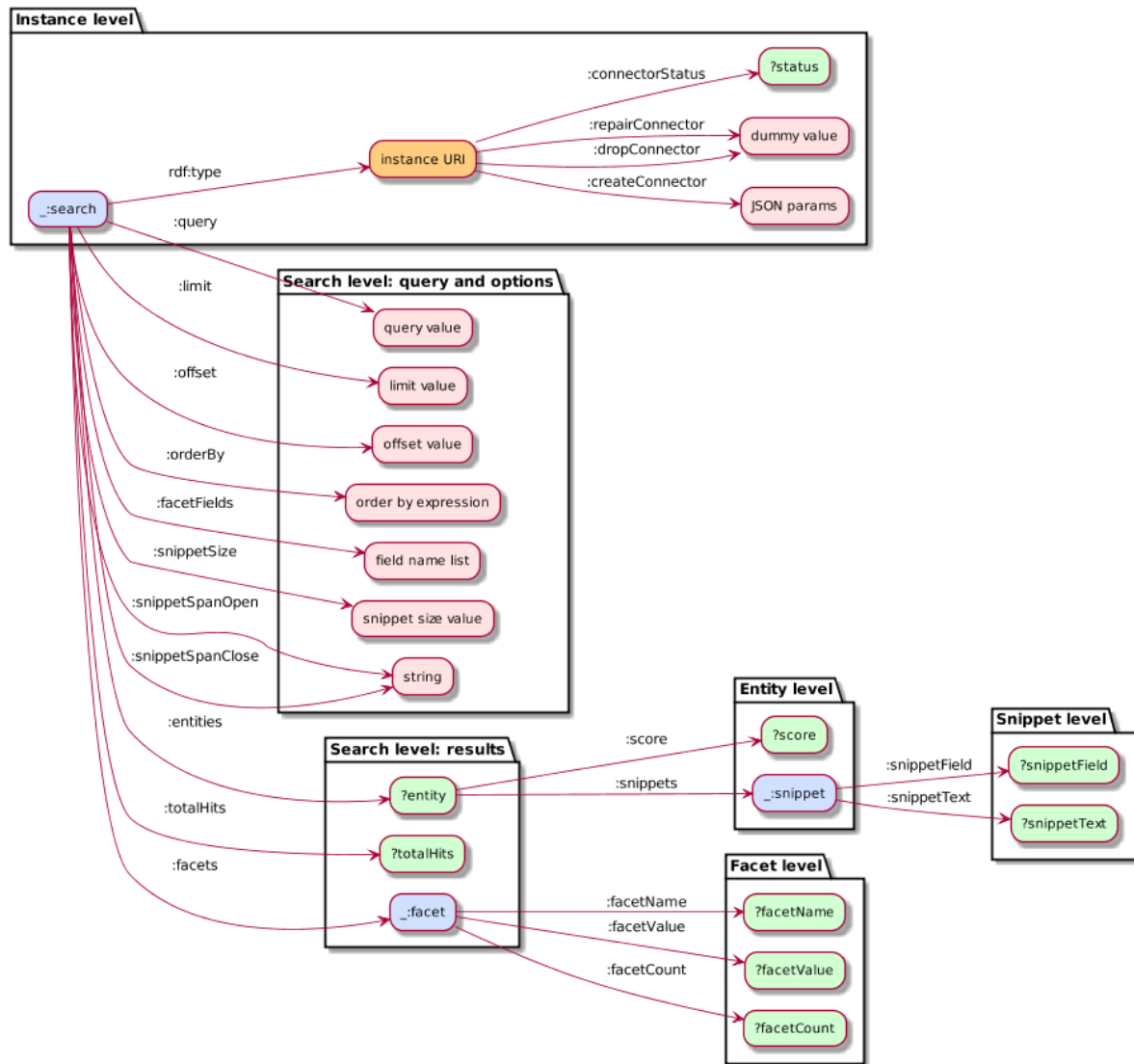
SELECT ?facetName ?facetValue ?facetCount {
  ?search a inst:my_index ;
    :facetFields "taggedWithLocation,taggedWithPerson" ;
    :facets _:f .
  _:f :facetName ?facetName ;
    :facetValue ?facetValue ;
    :facetCount ?facetCount .
}
```

If the filter was applied, you should get only :Berlin for taggedWithLocation and only :Einstein and :Mozart for taggedWithPerson:

?facetName	?facetValue	?facetCount
taggedWithLocation	http://www.ontotext.com/example2#Berlin	3
taggedWithPerson	http://www.ontotext.com/example2#Mozart	2
taggedWithPerson	http://www.ontotext.com/example2#Einstein	1

5.10.1.8 Overview of connector predicates

The following diagram shows a summary of all predicates that can administer (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 URIs in orange. The predicates are represented by labelled arrows.



5.10.1.9 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.

5.10.1.10 Upgrading from previous versions

Migrating from GraphDB 6.2 to 6.6

There are no new connector options in GraphDB 7.

The Lucene Connector in GraphDB 6.2 to 6.6 uses Lucene 4.x and the Lucene Connector in GraphDB 7 uses Lucene 5.x. GraphDB 7 can use connector instances created with GraphDB 6.2 to 6.6 with the following exception:

- Fields used for sorting (orderBy predicate) need to be declared with multivalued = false now. If you use orderBy you have to recreate your connector instances.

We recommend to drop any existing instances and recreate them to benefit from any performance improvements in Lucene 5.x even if you do not have any orderBy's in your queries.

Migrating from a pre-6.2 version

GraphDB prior to 6.2 shipped with version 3.x of the Lucene GraphDB Connector that had different options and slightly different behaviour and internals. Unfortunately, it is not possible to migrate existing connector instances automatically. To prevent any data loss, the Lucene GraphDB Connector will not initialise, if it detects an existing connector in the old format. The recommended way to migrate your existing instances is:

1. Backup the INSERT statement used to create the connector instance.
2. Drop the connector.
3. Deploy the new GraphDB version.
4. Modify the INSERT statement according to the changes described below.
5. Re-create the connector instance with the modified INSERT statement.

You might also need to change your queries to reflect any changes in field names or extra fields.

Changes in field configuration and synchronization

Prior to 6.2, a single field in the config could produce up to three individual fields on the Lucene side, based on the field options. For example, for the field “firstName”:

field	note
firstName	produced, if the option “index” was true; used explicitly in queries
_facet_firstName	produced, if the option “facet” was true; used implicitly for facet search
_sort_firstName	produced, if the option “sort” was true; used implicitly for ordering connector results

The current version always produces a single Lucene field per field definition in the configuration. This means that you have to create all appropriate fields based on your needs. See more in [List of creation parameters](#).

Tip: To mimic the functionality of the old _sort_fieldName fields, you can either create a non-analyzed [Copy fields](#) (for textual fields) or just use the normal field (for non-textual fields).

5.11 MongoDB Integration

What's in this document?

- [Overview and features](#)
- [Usage](#)
- [Setup and maintenance](#)
 - [Installing MongoDB](#)
 - [Creating an index](#)
 - [Upgrading an index](#)
 - [Deleting an index](#)

- *Loading sample data*
- *Querying MongoDB*
- *Multiple index calls in the same query*
- *Using aggregation functions*
- *Custom fields*
- *Authentication*

5.11.1 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 which sends a request to MongoDB then transforms the result to RDF model.

Each feature is described in detail below.

5.11.2 Usage

The steps for using MongoDB with GraphDB are:

- Installing MongoDB;
- Preparing and loading [JSON-LD](#) documents in MongoDB;
- 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_Bricaire_de_la_Dixmerie"
        },
        {
            "@id": "dbpedia:Bernard_Roberts"
        },
        {
            "@id": "dbpedia:Bartolomé_de_Medina"
        },
        {
            "@id": "dbpedia:Don_Bonker"
        },
        {
            "@id": "dbpedia:Cornel_Nistorescu"
        },
        {
            "@id": "dbpedia:Clete_Roberts"
        },
        {
            "@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 disbelieved might.",
    "cwork:thumbnail": {
        "@id": "bbct:1361611547"
    },
    "cwork:title": "Beckhoff greatly agitators constructed racquets industry restrain_
↪spews pitifully undertone stultification."
    ],
    "@id": "bbcc:1#id",
    "@context": {
        "bbcevent": "http://www.bbc.co.uk/ontologies/event/",
        "geo-pos": "http://www.w3.org/2003/01/geo/wgs84_pos#",
        "bbc": "http://www.bbc.co.uk/ontologies/bbc/",
        "time": "http://www.w3.org/2006/time#",
        "event": "http://purl.org/NET/c4dm/event.owl#",
        "music-ont": "http://purl.org/ontology/mo/",
        "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        "foaf": "http://xmlns.com/foaf/0.1/",
        "provenance": "http://www.bbc.co.uk/ontologies/provenance/",
        "owl": "http://www.w3.org/2002/07/owl#",
        "cms": "http://www.bbc.co.uk/ontologies/cms/",
        "news": "http://www.bbc.co.uk/ontologies/news/",
        "cnews": "http://www.bbc.co.uk/ontologies/news/cnews/",
        "cconcepts": "http://www.bbc.co.uk/ontologies/coreconcepts/",
        "dbp-prop": "http://dbpedia.org/property/",
        "geonames": "http://sws.geonames.org/",
        "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
        "domain": "http://www.bbc.co.uk/ontologies/domain/",
        "dbpedia": "http://dbpedia.org/resource/",
        "geo-ont": "http://www.geonames.org/ontology#",
        "bbc-pont": "http://purl.org/ontology/po/",
        "tagging": "http://www.bbc.co.uk/ontologies/tagging/",
        "sport": "http://www.bbc.co.uk/ontologies/sport/",
        "skosCore": "http://www.w3.org/2004/02/skos/core#",
        "dbp-ont": "http://dbpedia.org/ontology/",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "core": "http://www.bbc.co.uk/ontologies/coreconcepts/",
        "curric": "http://www.bbc.co.uk/ontologies/curriculum/",
        "skos": "http://www.w3.org/2004/02/skos/core#",
        "cwork": "http://www.bbc.co.uk/ontologies/creativework/",
        "fb": "http://rdf.freebase.com/ns/",
        "ot": "http://www.ontotext.com/",
        "ldbcspb": "http://www.ldbcouncil.org/spb#",
        "bbcd": "http://www.bbc.co.uk/document/",
        "bbcc": "http://www.bbc.co.uk/context/",
        "bbct": "http://www.bbc.co.uk/thumbnail/"
    }
}

```

- `_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.

5.11.3 Setup and maintenance

5.11.3.1 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 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](#).

5.11.3.2 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 let you connect to the database.

Below is a sample query of how to create a MongoDB index:

```
PREFIX mongodb: <http://www.ontotext.com/connectors/mongodb#>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
  inst:spb1000 :service "mongodb://localhost:27017" ;
    :database "ldb" ;
    :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.

5.11.3.3 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.

5.11.3.4 Deleting an index

Deletion of an index is done using the following query:

```
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
INSERT DATA {
    inst:spb1000 :drop _:b .
}
```

5.11.3.5 Loading sample data

Import provided cwork1000.json file with 1000 of CreativeWork documents in MongoDB database “ldbc” and “creativeWorks” collection.

```
mongoimport --db ldbc --collection creativeWorks --file cwork1000.json
```

5.11.3.6 Querying MongoDB

Below is a sample query which returns the dateModified for docs with the specific audience:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>

SELECT ?creativeWork ?modified WHERE {
    ?search a inst:spb1000 ;
        :find '{"@graph.cwork:audience.@id" : "cwork:NationalAudience"}' ;
        :entity ?entity .
    GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified .
    }
}
```


	creativeWork	modified
1	http://www.bbc.co.uk/things/7#id	"2011-05-24T20:58:30.144+03:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
3	http://www.bbc.co.uk/things/14#id	"2011-12-27T20:21:48.733+02:00"^^xsd:dateTime
4	http://www.bbc.co.uk/things/11#id	"2012-01-23T10:16:23.268+02:00"^^xsd:dateTime
5	http://www.bbc.co.uk/things/15#id	"2011-04-02T09:47:34.612+03:00"^^xsd:dateTime
6	http://www.bbc.co.uk/things/25#id	"2012-01-04T13:52:07.957+02:00"^^xsd:dateTime
7	http://www.bbc.co.uk/things/29#id	"2012-02-04T16:13:33.701+02:00"^^xsd:dateTime
8	http://www.bbc.co.uk/things/31#id	"2011-12-27T09:36:20.211+02:00"^^xsd:dateTime
9	http://www.bbc.co.uk/things/30#id	"2011-04-23T08:12:33.724+03:00"^^xsd:dateTime
10	http://www.bbc.co.uk/things/34#id	"2012-02-28T02:53:15.117+02:00"^^xsd:dateTime

In a query, use the exact values as in the docs. For example, if the full URIs are used instead of “cwork:NationalAudience” or “@graph.cwork:audience.@id” there wouldn’t 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()`)

5.11.3.7 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:

```

PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
  {
    SELECT ?creativeWork ?modified {
      ?search a inst:spb1000 ;
      :find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
      :entity ?creativeWork .
      GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified ;
      }
    }
  }
  UNION
  {
    SELECT ?creativeWork ?modified WHERE {
      ?search a inst:spb1000 ;
      :find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
      :entity ?entity .
      GRAPH inst:spb1000 {
        ?creativeWork cwork:dateModified ?modified ;
      }
    }
  }
}

```

Example 2:

```

PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
SELECT ?creativeWork ?modified WHERE {
  {
    ?search a inst:spb1000 ;
    :graph :search1 ;
    :find '{"@graph.@id" : "http://www.bbc.co.uk/things/1#id"}' ;
    :entity ?creativeWork .
    GRAPH :search1 {
      ?creativeWork cwork:dateModified ?modified ;
    }
  }
  UNION
  {
    ?search a inst:spb1000 ;
    :graph :search2 ;
    :find '{"@graph.@id" : "http://www.bbc.co.uk/things/2#id"}' ;
    :entity ?entity .
    GRAPH :search2 {
      ?creativeWork cwork:dateModified ?modified ;
    }
  }
}

```

Both examples return the same result.

	creativeWork	modified
1	http://www.bbc.co.uk/things/1#id	"2012-02-14T12:43:13.165+02:00"^^xsd:dateTime
2	http://www.bbc.co.uk/things/2#id	"2011-11-27T13:17:29.243+02:00"^^xsd:dateTime

5.11.3.8 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 inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?s ?o {
  ?search a inst:spb1000 ;
    :aggregate '["$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}],
    {"$addFields": {"@graph.cwork:graph.@id" : "$@id"}}]' ;
    :entity ?entity .
  GRAPH inst: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 inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?g1 ?g2 {
  ?search a inst:spb1000 ;
    :aggregate '["$match": {"@graph.@id": "http://www.bbc.co.uk/things/1#id"}],
    {"$addFields": {"@graph.inst:graph.@id" : "$@id"}}]' ;
    :entity ?entity .
  GRAPH inst:spb1000 {
    OPTIONAL {
      ?s inst:graph ?g1 .
    }
    ?s <inst:graph> ?g2 .
  }
}
```

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.

5.11.3.9 Custom fields

Example:

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX inst: <http://www.ontotext.com/connectors/mongodb/instance#>
PREFIX : <http://www.ontotext.com/connectors/mongodb#>
```

```

SELECT ?size ?halfSize {
  ?search a inst:spb1000 ;
    :aggregate '''[{"$match": {"@graph.@type": "cwork:NewsItem"}},
      {"$count": "size"},
      {"$project": {"custom.size": "$size", "custom.halfSize": {"$divide": ["$size", 2]}}}]
  ↪''';
  :entity ?entity .
  GRAPH inst:spb1000 {
    ?s inst:size ?size ;
    inst:halfSize ?halfSize .
  }
}

```

	size	halfSize
1	342	171.0

The values are projected as child elements of a custom node. After JSON-LD is taken from MongoDB, a pre-processing 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.

5.11.3.10 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.

5.12 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, 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`.

5.12.1 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:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {

```

```

?book rdfs:label "The Hitchhiker's Guide to the Galaxy" ;
  books:author ?author .

SERVICE <repository:authors> {
  ?author rdfs:label ?authorName
}

```

5.12.2 Parameters

There are two parameters that control how the federated part of the query is executed:

infer (boolean) Controls if inferred statements are included. True by default.

When this value is set to false, it is equivalent to adding FROM <http://www.ontotext.com/explicit> to the federated query.

sameAs (boolean) Controls if statements are expanded over owl:sameAs. True by default.

When this value is set to false, it is equivalent to adding FROM <http://www.ontotext.com/disable-sameAs> to the federated query.

To set a parameter put a comma after the special URL referring to the internal repository, then the parameter name, an equals sign and finally the value of the parameter. If you need to set more than one parameter put another comma, parameter name, equals sign and value.

Some examples:

repository:NNN,infer=false Turns off inference and inferred statements are not included in the results.

repository:NNN,sameAs=false Turns off the expansion of statements over owl:sameAs and they are not included in the results.

repository:NNN,infer=false,sameAs=false Turns off the inferred statements and they are not included in the results.

Turns off the expansion of statements over owl:sameAs and they are not included in the results.

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:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX books: <http://example.com/books/>

SELECT ?authorName WHERE {
  ?book rdfs:label "The Hitchhiker's Guide to the Galaxy" ;
    books:author ?author .

  SERVICE <repository:authors,infer=false,sameAs=false> {
    ?author rdfs:label ?authorName
  }
}

```

5.13 GraphDB dev guide

5.13.1 Storage

What's in this document?

- *What is GraphDB's persistence strategy*
- *GraphDB's indexing options*
 - *Transaction control*
 - *Predicate lists*
 - *Context index*
 - *Literal index*
 - *Handling of explicit and implicit statements*

5.13.1.1 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 indices available, all of which apply to all triples, whether explicit or implicit. These indices are maintained automatically.

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

- handling of billions of statements under reasonable RAM constraints;
- query optimisation;
- transaction management.

GraphDB maintains two main indices 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.

5.13.1.2 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 indices 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 initialisation – 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:

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

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 indices, no other transaction must be allowed to change anything until the first one is either committed or rolled back;

Note: GraphDB performs materialisation, 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 1000) 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., `FILTER(?x = 100 && ?y <= 5 && ?start > "2001-01-01"^^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"^^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-optimiser 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:

```
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 example shows how to retrieve only implicit statements:

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

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

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

SPARQL query evaluation

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

Tip: To learn more, see [Query behaviour](#).

5.13.2 Notifications

What's in this document?

- *What are GraphDB local notifications*
 - *How to register for local notifications*
- *What are GraphDB remote notifications*
 - *How to use remote notifications*

5.13.2.1 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:

```
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.

5.13.2.2 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 initialised in the same JVM that runs the example (local repository):

```
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

// stop listening for this pattern
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 initialised differently:

```
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.

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.

5.13.3 Query behaviour

What's in this document?

- *What are named graphs*
 - *The default SPARQL dataset*
- *How to manage explicit and implicit statements*
- *How to query explicit and implicit statements*
- *How to specify the dataset programmatically*
- *How to access internal identifiers for entities*
 - *Examples*
- *How to use RDF4J 'direct hierarchy' vocabulary*
- *Other special GraphDB query behaviour*

5.13.3.1 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:

Query	Bindings
SELECT * { ?s ?p ?o }	?s=ex:x ?p=ex:y ?o=ex:z
SELECT * { GRAPH ?g { ?s ?p ?o } }	?s=ex:x ?p=ex:y ?o=ex:z ?g=ex:g

In other words, the triple `ex:x ex:y ex:z` will appear to be in both the default graph and the named graph `ex:g`.

There are two reasons for this behaviour:

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 } }`.

5.13.3.2 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.

5.13.3.3 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.

5.13.3.4 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).

```
DatasetImpl dataset = new DatasetImpl();
dataset.addDefaultGraph(null);
dataset.addNamedGraph(valueFactory.createURI("http://example.com/g1"));
```

This dataset can then be passed to queries or updates, e.g.:

```
TupleQuery query = connection.prepareTupleQuery(QueryLanguage.SPARQL, queryString);
query.setDataset(dataset);
```

5.13.3.5 How to access internal identifiers for entities

Internally, GraphDB uses integer identifiers (IDs) to index all entities (URIs, blank nodes and literals). Statement indices 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	<code><http://www.ontotext.com/owlim/entity#id></code>
Description	A map between an entity and an internal ID
Example	Select all entities and their IDs: <pre>PREFIX ent: <http://www.ontotext.com/owlim/entity#> SELECT * WHERE { ?s ent:id ?id } ORDER BY ?id</pre>
Function	<code><http://www.ontotext.com/owlim/entity#id></code>
Description	Return an entity's internal ID
Example	Select all statements and order them by the internal ID of the object values: <pre>PREFIX ent: <http://www.ontotext.com/owlim/entity#> SELECT * WHERE { ?s ?p ?o . } order by ent:id(?o)</pre>

Examples

- Enumerate all entities and bind the nodes to ?s and their IDs to ?id, order by ?id:

```
select * where {  
  ?s <http://www.ontotext.com/owlim/entity#id> ?id  
} order by ?id
```

- Enumerate all non-literals and bind the nodes to ?s and their IDs to ?id, order by ?id:

```
SELECT * WHERE {  
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .  
  FILTER (!isLiteral(?s)) .  
} ORDER BY ?id
```

- Find the internal IDs of subjects of statements with specific predicate and object values:

```
SELECT * WHERE {  
  ?s <http://test.org#Pred1> "A literal".  
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .  
} ORDER BY ?id
```

- Find all statements where the object has the given internal ID by using an explicit, untyped value as the ID (the "115" is used as object in the second statement pattern):

```
SELECT * WHERE {  
  ?s ?p ?o.  
  ?o <http://www.ontotext.com/owlim/entity#id> "115" .  
}
```

- As above, but using an xsd:long datatype for the constant within a FILTER condition:

```
SELECT * WHERE {  
  ?s ?p ?o.  
  ?o <http://www.ontotext.com/owlim/entity#id> ?id .  
  FILTER (?id="115"^^<http://www.w3.org/2001/XMLSchema#long>) .  
} ORDER BY ?o
```

- Find the internal IDs of subject and object entities for all statements:

```
SELECT * WHERE {  
  ?s ?p ?o.  
  ?s <http://www.ontotext.com/owlim/entity#id> ?ids.  
  ?o <http://www.ontotext.com/owlim/entity#id> ?ido.  
}
```

- Retrieve all statements where the ID of the subject is equal to "115"^^xsd:long, by providing an internal ID value within a filter expression:

```
SELECT * WHERE {  
  ?s ?p ?o.  
  FILTER ((<http://www.ontotext.com/owlim/entity#id>(?s))  
          = "115"^^<http://www.w3.org/2001/XMLSchema#long>).  
}
```

- Retrieve all statements where the string-ised ID of the subject is equal to "115", by providing an internal ID value within a filter expression:

```
SELECT * WHERE {  
  ?s ?p ?o.  
  FILTER (str( <http://www.ontotext.com/owlim/entity#id>(?s) ) = "115").  
}
```


5.13.3.6 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#>`

Predicate	Definition
A <code>sesame:directSubClassOf</code> 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 <code>sesame:directSubPropertyOf</code> 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 <code>sesame:directType</code> 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.

5.13.3.7 Other special GraphDB query behaviour

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.

5.13.4 Retain BIND position special graph

The default behavior of the GraphDB *query optimiser* 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:

```
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):

```
SELECT ?r {  
  ?q <urn:pp1> ?x .  
  ?q <urn:pp2> ?y .  
  BIND (?x + ?y + ?z AS ?r) .  
  ?q <urn:pp3> ?z .  
}
```

the ‘correct’ result would be:

```
1 result: r=UNDEF
```

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:pp2> ?y .  
  ?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 optimiser 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:

```
<http://www.ontotext.com/retain-bind-position>.
```

In this case, the optimiser 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 .  
  ?q <urn:pp2> ?y .  
  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"
```

5.13.5 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 1000, but it can be controlled by using the *graphdb.engine.min-replace-graph-tx-size* configuration parameter.

The algorithm has the following steps:

- Check transaction contents. If the transaction includes a graph replacement and is of sufficient size, proceed.
- 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.
- While processing transaction statements for insertion, if their context (graph) matches an identifier from the list, store them inside a tracker.
- While clearing the graph to be replaced, if it is not mentioned in the tracker, directly delete all its contents.
- If a graph is mentioned in the tracker, iterate over its triples.
- Triples in the replacement graph that are also in the tracker are preserved. Otherwise, they are deleted.

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

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

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

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

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

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

Note: The replacement optimization described here applies to all forms of transactions. 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>`

5.14 Experimental features

5.14.1 SPARQL-MM support

What's in this document?

- *Usage examples*
 - *Temporal relations*
 - *Temporal aggregation*
 - *Temporal accessors*
 - *Spatial relations*
 - *Spatial aggregation*
 - *General relation*
 - *General aggregation*
 - *General accessor*

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.

5.14.1.1 Usage examples**Temporal relations**

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm:precedes(?f1,?f2)
} ORDER BY ?t1 ?t2
```

Temporal aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:temporalIntermediate(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

Temporal accessors

```
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 WHERE {
    ?f1 a ma:MediaFragment.
} ORDER BY mm:duration(?f1)
```

Spatial relations

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm:rightBeside(?f1,?f2)
} ORDER BY ?t1 ?t2
```

Spatial aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:spatialIntersection(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

General relation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?t1 ?t2 WHERE {
    ?f1 rdfs:label ?t1.
    ?f2 rdfs:label ?t2.
    FILTER mm>equals(?f1,?f2)
} ORDER BY ?t1 ?t2
```

General aggregation

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?f1 ?f2 (mm:boundingBox(?f1,?f2) AS ?box) WHERE {
    ?f1 rdfs:label "a".
    ?f2 rdfs:label "b".
}
```

General accessor

```
PREFIX ma: <http://www.w3.org/ns/ma-ont#>
PREFIX mm: <http://linkedmultimedia.org/sparql-mm/ns/2.0.0/function#>

SELECT ?pixelURI WHERE {
    ?f1 ma:hasFragment ?f1.
    BIND (mm:toPixel(?f1) AS ?pixelURI)
} ORDER BY ?t1 ?t2
```

Tip: For more information, see:

- The SPARQL-MM Specification
 - List of SPARQL-MM functions
-

5.14.2 Provenance plugin

What's in this document?

- *When to use the Provenance plugin*
- *Predicates*
- *Enabling the plugin*
- *Using the plugin - examples*

5.14.2.1 When to use the Provenance plugin

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, the GraphDB's forward-chaining inferencer materialises 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.

5.14.2.2 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/>`:

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

5.14.2.3 Enabling the plugin

The plugin is disabled by default.

1. Start the plugin by adding the parameter:

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

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

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

5.14.2.4 Using the plugin - examples

1. Copy the TRIG file in the *Import -> RDF -> Text area* of the workbench:

```
@prefix food: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/food#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix vin: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

food:Ontology {

  food:Fruit a owl:Class ;
    rdfs:label "fruit"@en ;
    rdfs:comment "In botany, a fruit is the seed-bearing structure in flowering plants_
    ↪formed from the ovary after flowering".

  food:Grape rdfs:label "grape"@en ;
    rdfs:comment "A grape is a fruiting berry of the deciduous woody vines of the botanical_
    ↪genus Vitis";
    rdfs:subClassOf food:Fruit.
}

vin:Ontology {

  vin:WineGrape rdfs:label "wine grape"@en ;
    rdfs:comment "Grape used for the wine production";
    rdfs:subClassOf food:Grape.

  vin:RedWineGrape rdfs:label "white wine"@en;
    rdfs:comment "Red grape used for wine production";
    rdfs:subClassOf vin:WineGrape.

  vin:CabernetSauvignon rdfs:label "Cabernet Sauvignon"@en ;
    rdfs:comment "Cabernet Sauvignon is one of the world's most widely recognized red wine_
    ↪grape varieties";
    rdfs:subClassOf vin:RedWineGrape.
}
```

2. **Example 1:** Return all explicit and implicit statements

This example returns all explicit and implicit statements derived from the `vin:Ontology` graph. The `?ctx` variable binds a new graph `pr:derivedFrom` the `vin:Ontology` graph, which includes all its implicit and explicit statements.

```
PREFIX pr: <http://www.ontotext.com/provenance/>
PREFIX vin: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>

CONSTRUCT {
  ?subject ?predicate ?object
}
WHERE {
  vin:Ontology pr:derivedFrom ?ctx .
  ?ctx pr:subject ?subject .
  ?ctx pr:predicate ?predicate .
  ?ctx pr:object ?object .
}
```

The result set will not include statements in which `vin:WineGrape` is `food:Grape` or `food:Fruit`.

3. **Example 2:** Return only implicit statements

The query below extends the previous example by excluding the explicit statements. It returns only the

implicit statements materialised from vin:Ontology graph:

```
PREFIX pr: <http://www.ontotext.com/provenance/>
PREFIX vin: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>

CONSTRUCT {
  ?subject ?predicate ?object
}
WHERE {
  vin:Ontology pr:derivedFrom ?ctx .
  ?ctx pr:subject ?subject .
  ?ctx pr:predicate ?predicate .
  ?ctx pr:object ?object .
  MINUS {
    GRAPH vin:Ontology {
      ?subject ?predicate ?object
    }
  }
}
```

4. Example 3: Return all implicit statements from multiple graphs

The plugin accepts multiple graphs provided with a value keyword. The example returns all implicit statements derived from the vin:Ontology and food:Ontology graphs:

```
PREFIX pr: <http://www.ontotext.com/provenance/>
PREFIX vin: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
PREFIX food: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/food#>

CONSTRUCT {
  ?subject ?predicate ?object
}
WHERE {
  VALUES ?graph {
    food:Ontology vin:Ontology
  }
  ?graph pr:derivedFrom ?ctx .
  ?ctx pr:subject ?subject .
  ?ctx pr:predicate ?predicate .
  ?ctx pr:object ?object .
  MINUS {
    GRAPH vin:Ontology {
      ?subject ?predicate ?object
    }
  }
}
```

5.14.3 Nested repositories

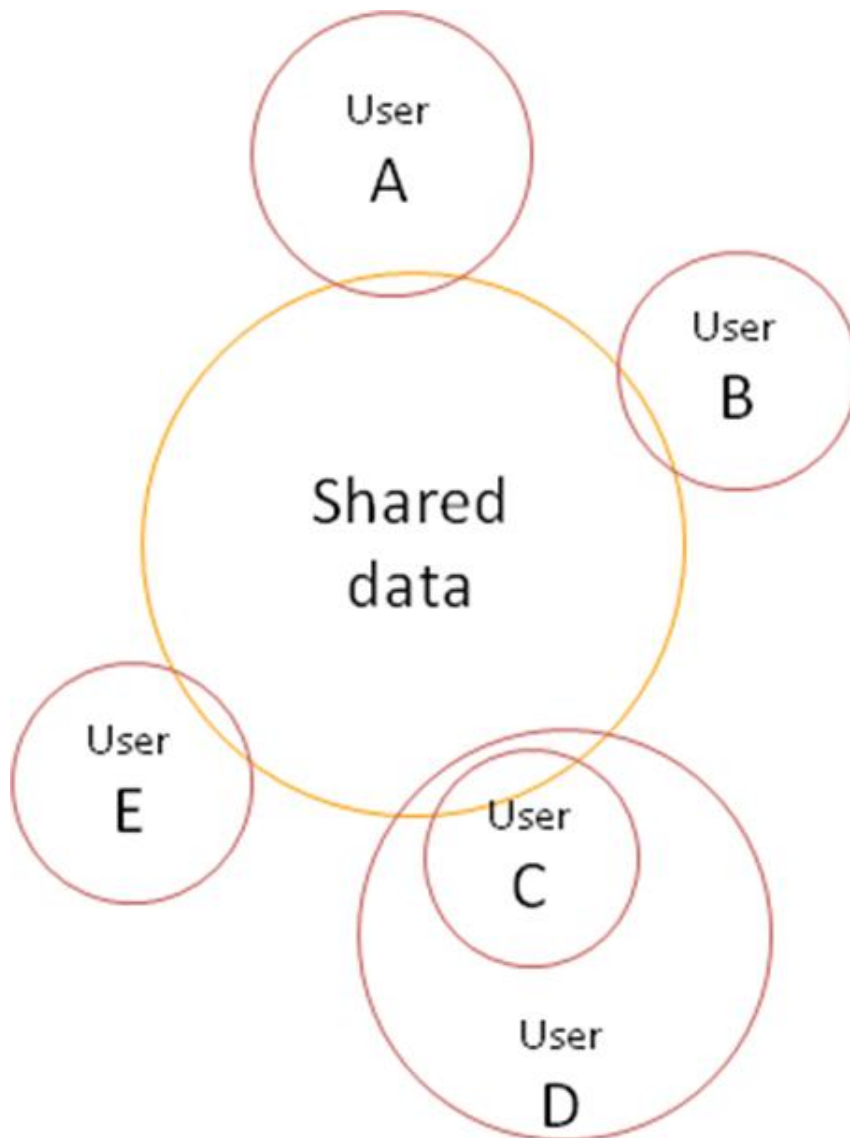
What's in this document?

- *What are nested repositories*
- *Inference, indexing and queries*
- *Configuration*
- *Initialisation and shut down*

5.14.3.1 What are nested repositories

Nested repositories is a technique for sharing RDF data between multiple GraphDB repositories. It is most useful when several logically independent repositories need to make use of a large (reference) dataset, e.g., a combination of one or more LOD datasets (GeoNames, DBpedia, MusicBrainz, etc.), but where each repository adds its own specific data. This mechanism allows the data in the common repository to be logically included, or ‘nested’, within other repositories that extend it. A repository that is nested in another repository (possibly into more than one other repository) is called a ‘parent repository’ while a repository that nests a parent repository is called a ‘child repository’. The RDF data in the common repository is combined with the data in each child repository for inference purposes. Changes in the common repository are reflected across all child repositories and inferences are maintained to be logically consistent.

Results for queries against a child repository are computed from the contents of the child repository, as well as the nested repository. The following diagram illustrates the nested repositories concept:



Note: When two child repositories extend the same nested repository, they remain logically separate. Only changes made to the common nested repository will affect any child repositories.

5.14.3.2 Inference, indexing and queries

A child repository ignores all values for its *ruleset configuration parameter* and automatically uses the same ruleset as its parent repository. Child repositories compute inferences based on the union of the explicit data stored in the child and parent repositories. Changes to either parent or child cause the set of inferred statements in the child to be updated.

Note: The child repository must be initialised (running) when updates to the parent repository take place, otherwise the child can become logically inconsistent.

When a parent repository is updated, before its transaction is committed, it updates every connected child repository by a set of statement INSERT/DELETE operations. When a child repository is updated, any new resources are recorded in the parent dictionary so that the same resource is indexed in the sibling child repositories using the same internal identifier.

Note: A current limitation on the implementation is that no updates using the *owl:sameAs* predicate are permitted.

Queries executed on a child repository should perform almost as well as queries executed against a repository containing all the data (from both parent and child repositories).

5.14.3.3 Configuration

Both parent and child repositories must be deployed using Tomcat and they must be deployed to the same instance on the same machine (same JVM).

Repositories that are configured to use the nesting mechanism must be created using specific RDF4J SAIL types:

- `owlim:ParentSail` - for parent (shared) repositories;
- `owlim:ChildSail` - for child repositories that extend parent repositories.

(Where the `owlim` namespace above maps to `http://www.ontotext.com/tree/owlim#`.)

Additional configuration parameters:

- `owlim:id` is used in the parent configuration to provide a nesting name;
- `owlim:parent-id` is used in the child configurations to identify the parent repository.

Once created, a child repository must not be reconfigured to use a different parent repository as this leads to inconsistent data.

Note: When setting up several GraphDB instances to run in the same Java Virtual Machine, i.e., the JVM used to host Tomcat, make sure that the configured memory settings take into account the other repositories. For example, if setting up 3 GraphDB instances, configure them as though each of them had only one third of the total Java heap space available.

5.14.3.4 Initialisation and shut down

To initialise nested repositories correctly, start the parent repository followed by each of its children.

As long as no further updates occur, the shutdown sequence is not strictly defined. However, we recommend that you shut down the children first followed by the parent.

5.14.4 LVM-based backup and replication

In essence, the Linux Logical Volume Management (LVM)-based Backup and Replication uses shell scripts to find out the logical volume and volume group where the repository storage folder resides and then creates a filesystem snapshot. Once the snapshot is created, the repository is available for reads and writes while the maintenance operation is still in-progress. When it finishes, the snapshot is removed and the changes are merged back to the filesystem.

What's in this document?

- *Prerequisites*
- *How it works*
- *Some further notes*

5.14.4.1 Prerequisites

- Linux OS;
- The system property (JVM's -D) named `lvm-scripts` should point to the folder with the above scripts;
- The folder you are about to backup or use for replication contains a file named `owlim.properties`;
- That folder DOES NOT HAVE a file named `lock`.

All of the above mean that the repository storage is 'ready' for maintenance.

5.14.4.2 How it works

By default, the LVM-based Backup and Replication feature is disabled.

To enable it:

1. Get the scripts located in the `lvmscripts` folder of the distribution.
2. Place them on each of the workers in a chosen folder.
3. Set the system property (JVM's -D) named `lvm-scripts`, e.g., `-Dlvm-scripts=<folder-with-the-scripts>`, to point to the folder with the scripts.

Note: GraphDB checks if the folder contains scripts named: `create-snapshot.sh`, `release-snapshot.sh`, and `locatelvm.sh`. This is done the first time you try to get the repository storage folder contents. For example, when you need to do backup or to perform full-replication.

GraphDB executes the script `locatelvm.sh` with a single parameter, which is the pathname of the storage folder from where you want to transfer the data (either to perform backup or to replicate it to another node). While invoking it, GraphDB captures the script standard and error output streams in order to get the logical volume, volume group, and the storage location, relative to the volume's mount point.

GraphDB also checks the exit code of the script (MUST be 0) and fetches the locations by processing the script output, e.g., it must contain the logical volume (after, `lv=`), the volume group (`vg=`), and the relative path (`local=`) from the mount point of the folder supplied as a script argument.

If the storage folder is not located on a LVM2 managed volume, the script will fail with a different exit code (it relies on the exit code of the `lvs` command) and the whole operation will revert back to the 'classical' way of doing it (same as in the previous versions).

If it succeeds to find the volume group and the logical volume, the `create-snapshot.sh` script is executed, which then creates a snapshot named after the value of `$BACKUP` variable (see the `config.sh` script, which also defines

where the snapshot will be mounted). When the script is executed, the logical volume and volume groups are passed as environment variables, named LV and VG preset by GraphDB.

If it passes without any errors (script exit code = 0), the node is immediately initialised in order to be available for further operations (reads and writes).

The actual maintenance operation will now use the data from the 'backup' volume instead from where it is mounted.

When the data transfer completes (either with an error, canceled or successfully), GraphDB invokes the `.release-snapshot.sh` script, which unmounts the backup volume and removes it. This way, the data changes are merged back with the original volume.

5.14.4.3 Some further notes

The scripts rely on a root access to do 'mount', and also to create and remove snapshot volumes. The SUDO_ASKPASS variable is set to point to the `askpass.sh` script from the same folder. All commands that need privilege are executed using `sudo -A`, which invokes the command pointed by the SUDO_ASKPASS variable. The latter simply spits out the required password to its standard output. You have to alter the `askpass.sh` accordingly.

During the LVM-based maintenance session, GraphDB will create two additional files (zero size) in the scripts folder, named `snapshot-lock`, indicating that a session is started, and `snapshot-created`, indicating a successful completion of the `create-snapshot.sh` script. They are used to avoid other threads or processes interfering with the maintenance operation that has been initiated and is still in progress.

SECURITY

Database security refers to the collective measures used to protect and secure a database from illegitimate use and malicious threats and attacks.

Database security covers and enforces security on many aspects. This includes:

6.1 Authorization

Authorization is the process of mapping a user to a set of specific permissions. GraphDB implements Spring security for authorization. Spring security acts as an interceptor and filter. Each permission, therefore, can be treated as a set of intercepted urls that are grouped logically for ease of visualization.

6.1.1 User roles and permissions

GraphDB implements the minimal Role Based Access Control (RBAC1) model with role hierarchy support. The model defines 4 entities:

- **User** - every user regardless if it is a anonymous or authenticated user;
- **Session** - the current session connections; every session always works in the context of a specific user;
- **Roles** - a group of permissions associated with a role; the roles may be organized in hierarchies i.e. Admin role give you all other roles without explicitly defining them;
- **Permission** - gives rights to execute a specific operation.

The roles defined in GraphDB security model follow a hierarchy

Role name	Associated permissions
Admin	Perform any server operations i.e. the security never rejects an operation
Repo Manager	Full read and write permissions to all repositories.
User	Can save SPARQL queries, graph visualizations or user pecific server side settings (“Setup” > “My settings”)

Each user may only be either a regular user, an admin or a repository manager. However, users can have multiple READ/WRITE roles.

Permissions	User	Repo Manager	Admin
Granted read access to a repository	yes	yes	yes
Granted read/write access to a repository	yes	yes	yes
Read and write all repositories	no	yes	yes
Create, edit and delete repositories	no	yes	yes
Access monitoring	no	yes	yes
Manage Connectors	no	yes	yes
Manage Users and Access	no	no	yes
Manage the cluster	no	no	yes
Attach remote locations	no	no	yes
View system information	no	no	yes

6.1.2 Built-in users and roles

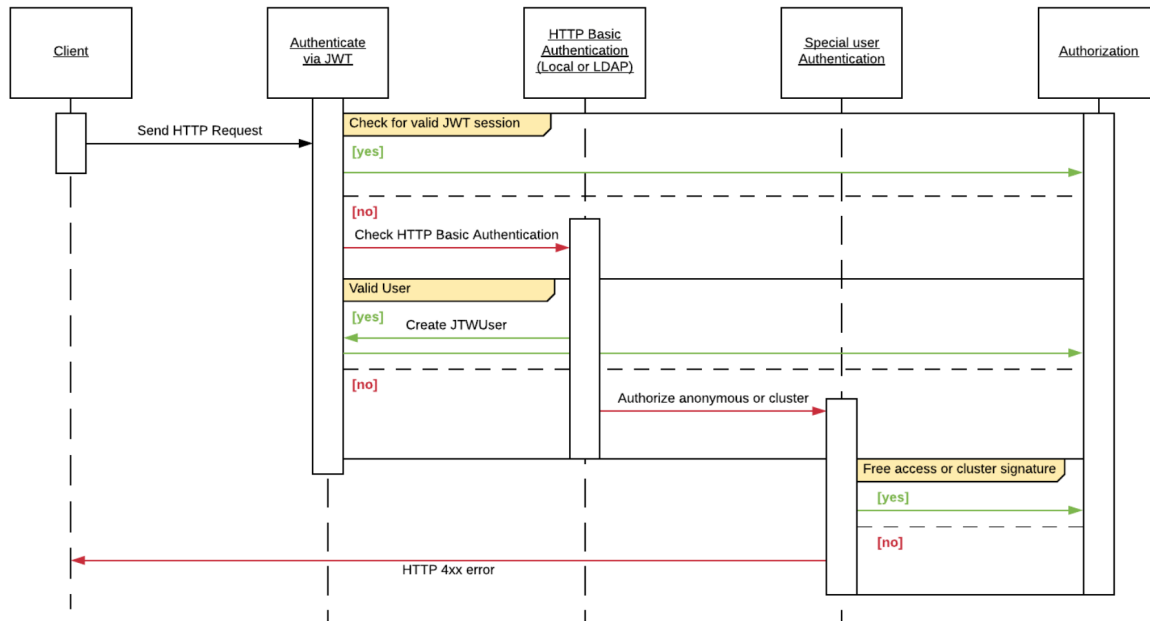
GraphDB has three special internal users that cannot be deleted and are needed for the functioning of the database.

User name	En-abled by default	Associated roles	Description
Admin	yes	ROLE_ADMIN	Default username and password in the local security database. Requires HTTP basic authentication.
<cluster user>	yes	ROLE_CLUSTER READ_REPO_* WRITE_REPO_*	The communication mechanisms in the cluster, where every request is signed with a special header “Authorization: GDB-Signature token”
<free access user>	no	none	The default user associated with anonymous user if anonymous access is enabled

6.2 Authentication

Whenever a client connects to GraphDB, a session is created. The session stores various contextual information about the connection. Each session is always associated with a single user. Authentication is the process of mapping this session to a specific user. Once the session is mapped to a user, a set of permissions can be associated with it, using authorization.

This diagram can be used as a quick reference for our security model:



GraphDB supports two authentication providers natively. In addition to this, it is possible to turn off GraphDB security completely and implement your own security layer instead.

6.2.1 Local

This is the default security access provider. Users stored inside the local database take precedence over LDAP users. For example, if both providers have a user called “leah”, the settings and roles for this user will be sourced from the local database.

The local database stores usernames, encrypted passwords, local user settings and remote LDAP user settings. Passwords are encrypted using the [SHA-256](#) algorithm. When a user tries to access the database, they will be met with a security challenge. If they can provide a valid username/password combination, a [JWT token](#) will be generated.

The local database is located in the settings.js file under the GraphDB work - workbench directory. By default this is `${graphdb.home}/work/workbench`. If you are worried about the security of this file, we recommend encrypting it (see [Encryption at rest](#)).

Note: JWT is serialized as “Authorization: GDB <token>” header in every request, so it is vulnerable to a man-in-the-middle attack. Everyone who intercepts the JWT can reuse the session. To prevent this, we recommend to always enable [encryption in transit](#).

Tokens are by default valid for 30 days. Within this period, each communication between the client and the server will be carried out only after the token has been validated. If the token has expired or is not present, the users will be met with a security challenge. You can configure this period via the `graphdb.auth.token.validity` parameter, changing its default value of 30 to a number of your choice.

Note: During the token validity period, if the password is changed the user would still have access to the server. However, if the user is removed the token would stop working.

Local authentication does not need to be configured.

6.2.2 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.

An optional identity provider. In the event that a given username/password combination cannot be validated by the local database, and a LDAP server has been configured, GraphDB security will attempt to fetch the user from the LDAP server.

Even with LDAP turned on, the local 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.

Unlike local authentication, LDAP needs to be configured in the graphdb.properties file, by passing the configuration details to the Java process or via the GDB_JAVA_OPTS parameter.

Note: The order of precedence for GraphDB configuration properties is: config file < GDB_JAVA_OPTS < command line supplied arguments.

LDAP is turned on by using the following setting:

```
graphdb.auth.module=ldap
```

When LDAP is turned on, the following security settings can be used to configure it:

Property	Values	Usage
graphdb.auth.ldap.url (required)	ldap:// <my-openldap-server> :389/<partition>	LDAP endpoint
graphdb.auth.ldap.user.search.base	<empty>	Query to identify the directory where all authenticated users are located.
graphdb.auth.ldap.user.search.filter (required)	(cn={0})	Matches the attribute to a GraphDB username.
graphdb.auth.ldap.role.search.base	<empty>	Query to identify the directory where roles/groups for authenticated users are located
graphdb.auth.ldap.role.search.filter (required)	(uniqueMember={0})	Authorize a user by matching the manner in which they are listed within the group.
graphdb.auth.ldap.role.search.attribute	cn (default)	The attribute to identify the common name
graphdb.auth.ldap.role.map.administrator (required)	my-group-name	Map a single LDAP group to GDB administrator role
graphdb.auth.ldap.role.map.repositoryManager	my-group-name	Map a single LDAP group to GDB repository manager role
graphdb.auth.ldap.role.map.repository.read.<my-repo>	my-group-name	Map a single LDAP group to GDB repository-specific read permissions
graphdb.auth.ldap.role.map.repository.write.<my-repo>	my-group-name	Map a single LDAP group to GDB repository-specific write permissions
graphdb.auth.ldap.repository.read.base	<empty>	Query to identify the directory where repository read groups for authenticated users are located
graphdb.auth.ldap.repository.read.filter	(uniqueMember={0})	Authorize a user by matching the manner in which they are listed within the group
graphdb.auth.ldap.repository.read.attribute	cn (default)	Specify the mapping of a GraphDB repository id to a LDAP attribute
graphdb.auth.ldap.repository.write.base	<empty>	Query to identify the directory where repository write groups for authenticated users are located.
graphdb.auth.ldap.repository.write.filter	(uniqueMember={0})	Authorize a user by matching the manner in which they are listed within the group
graphdb.auth.ldap.repository.write.attribute	cn (default)	Specify the mapping of a GraphDB repository id to a LDAP attribute

Note: Configuration changes happen only after restart.

Here is an example configuration:

```
# Turn on ldap authentication and configure the server.
graphdb.auth.ldap.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"
↳organisation.
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={0})
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

# All entries located under the "groups" organizational unit that have members (i.e., groups), will
↳be able to read repositories that share their common name.
graphdb.auth.ldap.repository.read.base = ou=groups
graphdb.auth.ldap.repository.read.filter = (member={0})
graphdb.auth.ldap.repository.read.attribute = cn

# All entries located under the "groups" organizational unit that have members (i.e., groups), will
↳be able to read and write to repositories that share their common name.
graphdb.auth.ldap.repository.write.base = ou=groups
graphdb.auth.ldap.repository.write.filter = (member={0})
graphdb.auth.ldap.repository.write.attribute = cn
```

6.2.3 Basic Authentication

Instead of using JWT, users can access GraphDB by passing valid base-64 encoded username:password combinations as a header, in the following fashion:

Authorization: Basic YWRtaW46cm9vdA==

Note: Basic Authentication is even more vulnerable to man-in-the-middle attacks than JWT! Anyone who intercepts your requests will be able to reuse your credentials indefinitely until you change them. Since the credentials are merely base-64 encoded, they will also get your username and password. This is why it is very important to always use *encryption in transit*.

Note: If you need to speed up a slow repository with enabled security when each request includes HTTP basic authentication, we recommend some *options here*.

6.3 Encryption

What's in this document?

- *Encryption in transit*
 - *Enable SSL/TLS*
 - *HTTPS in the Cluster*
- *Encryption at rest*

6.3.1 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.

6.3.1.1 Enable SSL/TLS

As GraphDB runs on embedded Tomcat server the security configuration is standard with a few exceptions. You can find the official Tomcat documentation on how to enable [SSL/TLS](#). Additional information on how to configure your GraphDB instance to use SSL/TLS could be found in the Configuration part of this document.

6.3.1.2 HTTPS in the Cluster

As there is a lot of traffic between the cluster nodes it is important that it is encrypted. In order to do so a few requirements should 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 URLs of the remote location (configured in *Setup -> Repositories -> Attach Remote Location*) should be using the HTTPS scheme.

The method of enabling SSL/TLS is already described in the upper section. There are no differences when setting up the node to be used as a cluster one. In order to achieve the certificate trust between the nodes you have a few options.

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 clients. 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 certificate is highly dependant on the CA.

Use Self-Signed certificates

The benefit is that you generate these certificates yourself and there is no need for somebody to sign them. However, the drawback is that by default the nodes will not trust the other nodes' certificates.

If you generate a separate self-signed certificate for each node in the cluster this certificate would have to be present in the Java Truststores of all other nodes in the cluster. You could do this by either adding the certificate to the default Java Truststore or specify 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, it is recommended that, instead of separate certificates for each node, you generate a single self-signed certificate and use it on all Cluster nodes. GraphDB extends the standard Java TrustManager so it will automatically trust its own certificate. This means that if all nodes in the cluster are using a shared certificate there would be no need to add it to the Truststore.

Another difference with 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 which will be using the certificate as Subject Alternative Names when generating the certificate (wildcards can be used as well).

Both options to trust your own certificate and to skip 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

6.3.2 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 noticeable performance impact hasn't been observed. However, please keep in mind that such may be present and is highly dependant on your specific use case.

6.4 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 to this, 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 which 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 require 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 parameter which configures this, `graphdb.audit.headers` 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 1024 if the `graphdb.audit.request.max.length` parameter is not set.

Note: Logs can be space-intensive, in particular if you toggle them to level 1 or 2, as described above.

7.1 Data modelling 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.

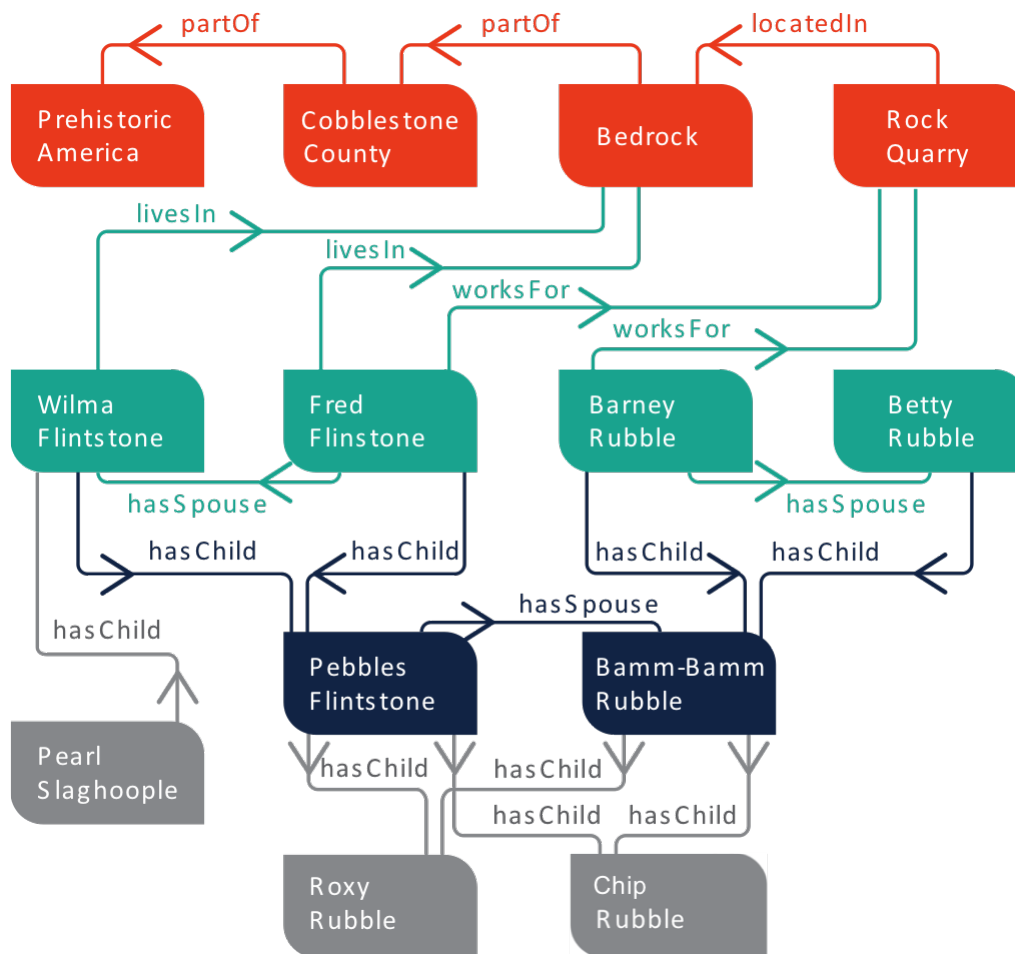
RDF consists of triples. These triples are based on an Entity Attribute Value (EAV) model, in which the subject is the entity, the predicate is the attribute, and the object is the value. Each triple has a unique identifier known as the Uniform Resource Identifier, or URI. URI's look like web page addresses. The parts of a triple, the subject, predicate, and object, represent links in a graph.

Example triples:

subject	predicate	object
:Fred	:hasSpouse	:Wilma
:Fred	:hasAge	25

In the first triple, “Fred hasSpouse Wilma”, Fred is the subject, hasSpouse is the predicate and Wilma is the object. Also, in the next triple, “Fred hasAge 25”, Fred is the subject, hasAge is the predicate and 25 is the object, or value.

Multiple triples link together to form an RDF model. The graph below describes the characters and relationships from the Flintstones television cartoon series. We can easily identify triples such as “WilmaFlintstone livesIn Bedrock” or “FredFlintstone livesIn Bedrock”. We now know that the Flintstones live in Bedrock, which is part of Cobblestone County in Prehistoric America.



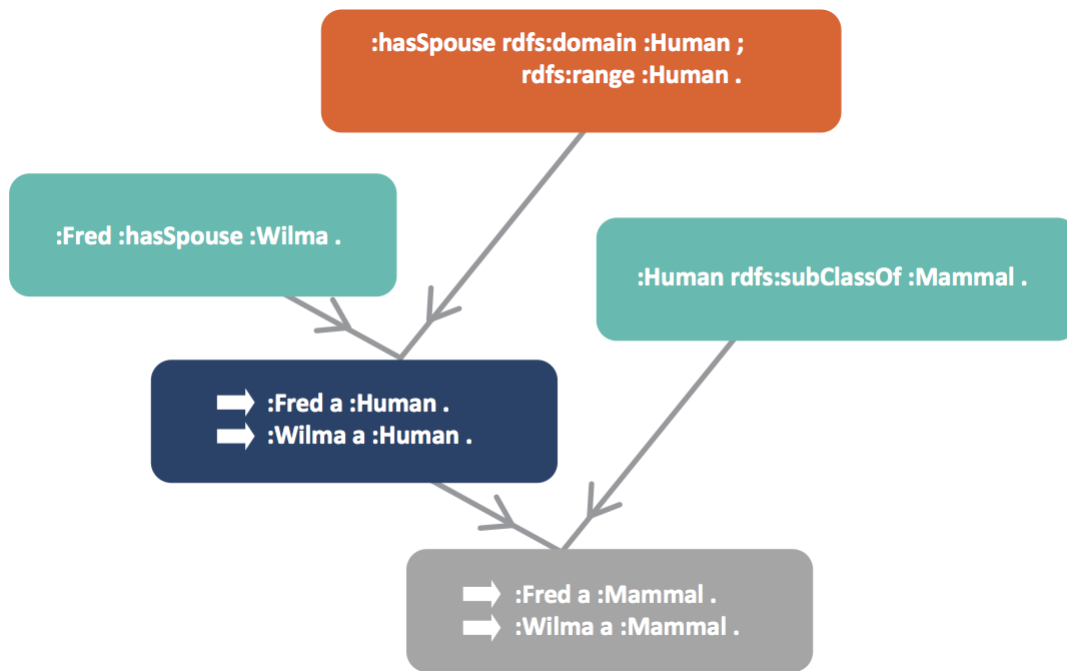
The rest of the triples in the Flintstones graph describe the characters' relations, such as `hasSpouse` or `hasChild`, as well as their occupational association (`worksFor`).

Fred Flintstone is married to Wilma and they have a child Pebbles. Fred works for the Rock Quarry company and Wilma's mother is Pearl Slaghoople. Pebbles Flintstone is married to Bamm-Bamm Rubble who is the child of Barney and Betty Rubble. Thus, as you can see, many triples form an RDF model.

7.1.2 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.

7.1.3 See also

You can also watch the video from GraphDB Fundamentals Module 1:

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 doesn't know the structure of the RDF data in the data source.
- `INSERT` adds triples to a graph,
- `DELETE` removes triples from a graph.

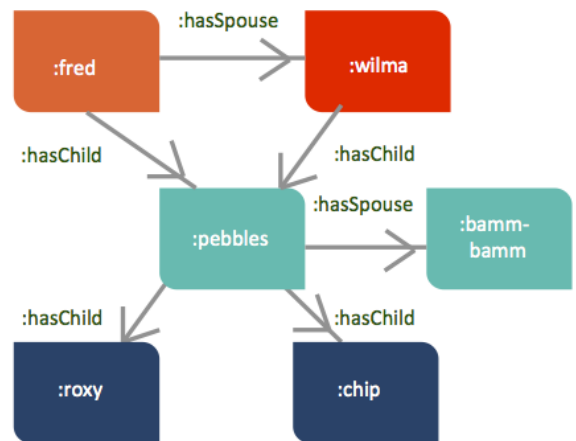
Let's use SPARQL, the query language for RDF graphs, to create a graph. To write the SPARQL query which creates an RDF graph, perform these steps:

First, define prefixes to URIs with the `PREFIX` keyword. In the example below, we set `bedrock` as the default namespace for the query.

Next, use `INSERT DATA` to signify you want to insert statements. Write the subject predicate object statements.

Finally, execute this query:

```
PREFIX : <http://bedrock/>
INSERT DATA {
  :fred    :hasSpouse :wilma .
  :fred    :hasChild  :pebbles .
  :wilma   :hasChild  :pebbles .
  :pebbles :hasSpouse :bamm-bamm ;
           :hasChild  :roxy, :chip .
}
```



As you can see in the example shown in the gray box, we wrote a query which included PREFIX, INSERT DATA, and several subject predicate object statements, which are:

Fred has spouse Wilma, Fred has child Pebbles, Wilma has child Pebbles, Pebbles has spouse Bamm-Bamm, and Pebbles has children Roxy and Chip.

Now, let's write a SPARQL query to access the RDF graph you just created.

First, define prefixes to URIs with the PREFIX keyword. As in the earlier example, we set bedrock as the default namespace for the query.

Next, use SELECT to signify you want to select certain information, and WHERE to signify your conditions, restrictions, and filters.

Finally, execute this query:

```
PREFIX : <http://bedrock/>
SELECT ?subject ?predicate ?object
WHERE { ?subject ?predicate ?object }
```

subject	predicate	object
:fred	:hasChild	:pebbles
:pebbles	:hasChild	:roxy
:pebbles	:hasChild	:chip
:wilma	:hasChild	:pebbles

As you can see in this example shown in the gray box, we wrote a SPARQL query which included PREFIX, SELECT, and WHERE. The red box displays the information which is returned in response to the written query. We can see the familial relationships between Fred, Pebbles, Wilma, Roxy, and Chip.

SPARQL is quite similar to SQL, however, unlike SQL which requires SQL schema and data in SQL tables, SPARQL can be used on graphs and does not need a schema to be defined initially.

In the following example, we will use SPARQL to find out if Fred has any grandchildren.

First, define prefixes to URIs with the PREFIX keyword.

Next, we use ASK to discover whether Fred has a grandchild, and WHERE to signify the conditions.


```

PREFIX : <http://bedrock/>
ASK
WHERE {
  :fred      :hasChild  ?child .
  ?child     :hasChild  ?grandChild .
}

```

→ YES

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:

```

PREFIX: <http://bedrock/>
SELECT ?grandChild
WHERE {
  :fred      :hasChild  ?child .
  ?child     :hasChild  ?grandChild .
}

```

grandChild

1. :roxy
2. :chip

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 ⓘ

Editor only Editor and results Results only ☐

Unnamed +

```

1 select * where {
2   ?s ?p ?o .
3 } limit 100
4

```

Run

Table Raw Response Pivot Table Google Chart Graph(beta) Download as ▶

Filter query results Showing results from 1 to 6. Query took 0.043 s.

	s	p	o
1	_:node1aflogvtx1	rdf:type	sys:RepositoryContext
2	_:node1aflogvtx2	sys:repositoryID	SYSTEM
3	_:node1aflogvtx3	sys:repositoryImpl	_:node1aflogvtx3
4	_:node1aflogvtx2	rdf:type	sys:Repository
5	_:node1aflogvtx2	http://www.w3.org/2000/01/rdf-schema#label	System configuration repository

keyboard shortcuts

7.2.3 See also

You can also watch the video from GraphDB Fundamentals Module 2:

7.3 Ontologies

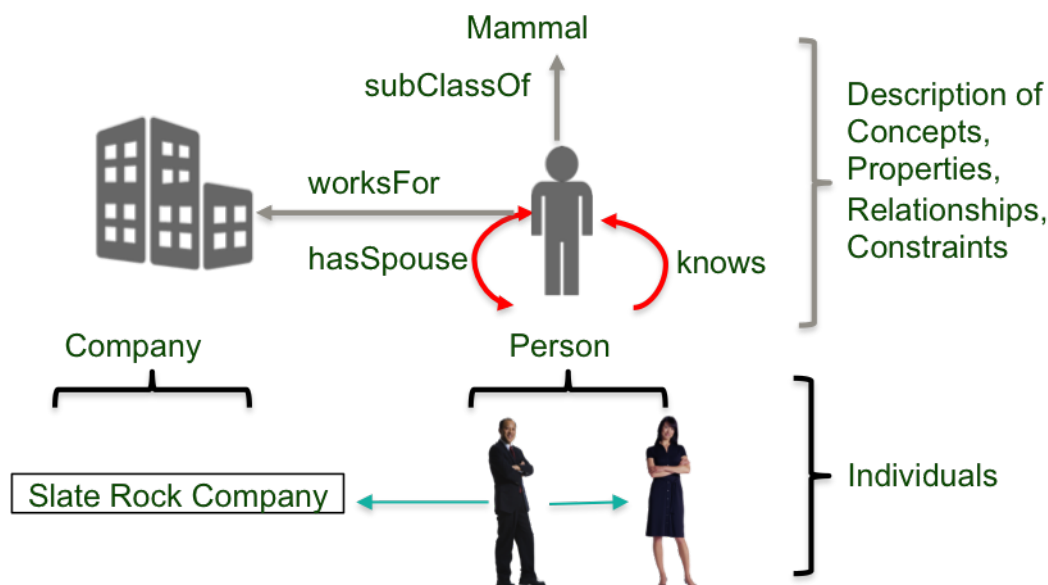
7.3.1 What is an ontology?

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.



7.3.2 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 meta data 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*.

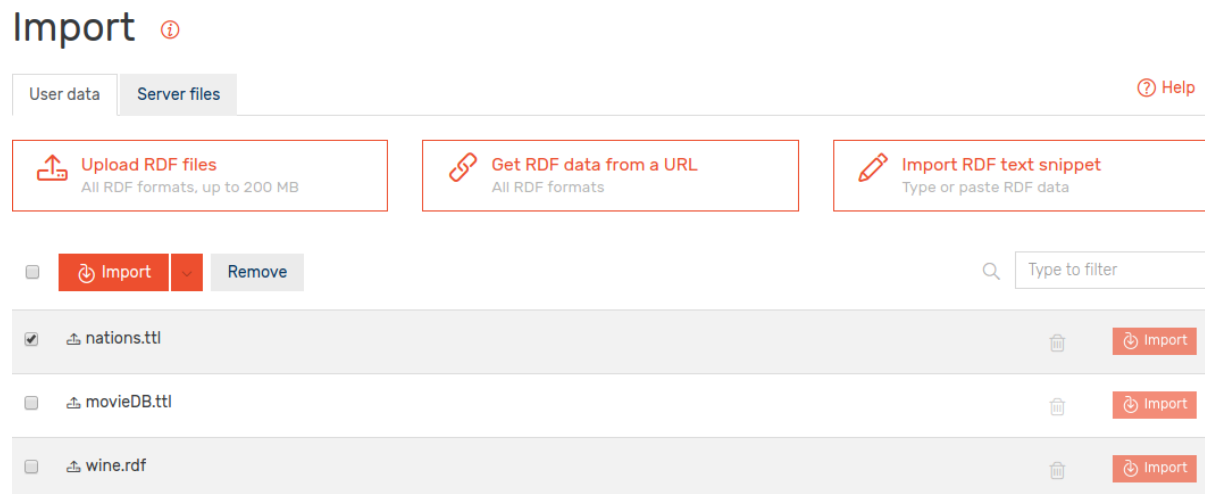
In addition, OWL offers more expressive class definitions, such as, class intersection, union, complement, disjointness and cardinality restrictions.

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.

7.3.3 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 -> RDF* view:



7.3.4 See also

You can also watch the video from GraphDB Fundamentals Module 3:

7.4 Inference

7.4.1 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.

7.4.2 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 modelled with RDF(S) and ontologies. GraphDB will use the data and the rules to infer more facts and thus produce a richer data set than the one you started with.

GraphDB can be configured via “rule-sets” – 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.

7.4.2.1 Standard rule-sets

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

To apply a rule-set, simply choose from the options in the pull-down list when configuring your repository as shown below through GraphDB Workbench:

Configure New Repository

Repository properties

Repository ID*	worker-test
Repository title	OWLIM-Enterprise worker node
Type	GRAPHDB-SE
Storage folder	storage
Ruleset	OWL-Horst (Optimized) No inference RDFS OWL-Horst OWL-Max OWL2-QL OWL2-RL RDFS (Optimized) OWL-Horst (Optimized) OWL-Max (Optimized) OWL2-QL (Optimized) OWL2-RL (Optimized)
Upload custom rule set (.pie)	
Base URL	
Entity index size	
Total cache memory	

7.4.2.2 Custom rule-sets

GraphDB also comes with support for custom rule-sets that allow for custom reasoning through the same performance optimised inference engine. The rule-sets are defined via .pie files.

To load custom rule-sets, simply point to the location of your .pie file as shown below:

Repository properties

Repository ID*	BOA
Repository title	REPO-BOA
Type	GRAPHDB-SE ▼
Storage folder	storage
Ruleset	OWL-Horst (Optimized) ▼
Upload custom rule set (.pie)	 No file choosen Select file

7.5 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.

7.5.1 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: <http://mvnrepository.com/artifact/com.ontotext.graphdb/graphdb-free-runtime>

7.5.2 Examples

The three examples below can be found under `examples/developer-getting-started` of the GraphDB distribution.

7.5.2.1 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
```

```

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
            // that can be traversed with the methods hasNext() and next()
            TupleQueryResult tupleQueryResult = tupleQuery.evaluate();
            while (tupleQueryResult.hasNext()) {
                // Each result is represented by a BindingSet, which corresponds to a result row
                BindingSet bindingSet = tupleQueryResult.next();

                // Each BindingSet contains one or more Bindings
                for (Binding binding : bindingSet) {
                    // Each Binding contains the variable name and the value for this result row
                    String name = binding.getName();
                    Value value = binding.getValue();

                    System.out.println(name + " = " + value);
                }

                // 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();
}
}

```

7.5.2.2 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 ." +
            "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 {" +
                "<%s> family:hasChild <%s>" +
                "}", personURI, childURI));

        // Committing the transaction persists the changes
        connection.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();
}
}
}

```

7.5.2.3 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.

```

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.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.repository.RepositoryException;
import org.eclipse.rdf4j.repository.base.RepositoryConnectionWrapper;
import org.eclipse.rdf4j.repository.config.RepositoryConfig;
import org.eclipse.rdf4j.repository.config.RepositoryConfigException;
import org.eclipse.rdf4j.repository.config.RepositoryConfigSchema;
import org.eclipse.rdf4j.repository.manager.LocalRepositoryManager;

```

```

import org.eclipse.rdf4j.repository.sail.config.SailRepositorySchema;
import org.eclipse.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.
     *
     * @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 if none should be overridden
     * @throws RDFParseException
     * @throws IOException
     * @throws RDFHandlerException
     * @throws GraphUtilException
     * @throws RepositoryConfigException
     * @throws RepositoryException
     */
    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, null)).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());
            graph.remove(configNode, key, null);
            graph.add(configNode, key, value);
        }
    }

    RepositoryConfig repositoryConfig = RepositoryConfig.create(graph, repositoryNode);

    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.

7.6 Extending GraphDB Workbench

What's in this document?

- *Clone, download, and run GraphDB Workbench*
- *Add your own page and controller*
- *Add repository checks*
- *Repository setup*
- *Select departure and destination airport*
- *Find the paths between the selected airports*
- *Visualize results*
- *Add status message*

GraphDB Workbench now becomes 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.

7.6.1 Clone, download, and run GraphDB Workbench

Download and run GraphDB 9.0 on the default port 7200.

Clone the [GraphDB Workbench](#) project from GitHub.

Enter the project directory and execute `npm install` in order to install all necessary dependencies locally.

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/`.

7.6.2 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` import the controller:

```
'angular/graphexplore/controllers/paths.controller',
```

add it to the route provider:

```
.when('/paths', {
  templateUrl: 'pages/paths.html',
  controller: 'GraphPathsCtrl',
  title: 'Graph Paths',
  helpInfo: 'Find all paths in a graph.',
});
```

and register it in the menu:

```
$menuItemProvider.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. In the `paths.controller.js` that you created, add:

```
define([
  'angular/core/services',
  'lib/common/d3-utils'
],
function (require, D3) {
  angular
    .module('graphdb.framework.graphexplore.controllers.paths', [
```

```

        'toastr',
        'ui.bootstrap',
    ])
    .controller('GraphPathsCtrl', GraphPathsCtrl);

    GraphPathsCtrl.$inject = ["$scope", "$rootScope", "$repositories", "toastr", "$timeout", "
↪$http", "ClassInstanceDetailsService", "AutocompleteService", "$q", "$location"];

    function GraphPathsCtrl($scope, $rootScope, $repositories, toastr, $timeout, $http,
↪ClassInstanceDetailsService, AutocompleteService, $q, $location) {

        }
    }
};

```

and register the module in `src/js/angular/graphexplore/modules.js`

```
'graphdb.framework.graphexplore.controllers.paths',
```

Now your controller and page are ready to be filled with content.

7.6.3 Add repository checks

In your page, you need a repository with data in it. Like most views in GraphDB, you need to have a repository set. The template that most of the pages use is similar to this, where the `repository-is-set` div is where you put your html. Error handling related to repository errors is added for you.

```

<div class="container-fluid">
  <h1>
    {{title}}
    <span class="btn btn-link"
      popover-template="'js/angular/templates/titlePopoverTemplate.html'"
      popover-trigger="mouseenter"
      popover-placement="bottom-right"
      popover-append-to-body="true"><span class="icon-info"></span></span>
  </h1>
  <div core-errors></div>
  <div system-repo-warning></div>
  <div class="alert alert-danger" ng-show="repositoryError">
    <p>The currently selected repository cannot be used for queries due to an error:</p>
    <p>{{repositoryError}}</p>
  </div>

  <div id="repository-is-set" ng-show="getActiveRepository() && !isLoadingLocation() &&
↪hasActiveLocation() && 'SYSTEM' !== getActiveRepository()">
    {{getActiveRepository()}}
  </div>
</div>

```

You need to define the functions this snippet depends on in your `paths.controller.js`. They use the repository service that you imported in the controller definition.

```

$scope.getActiveRepository = function () {
  return $repositories.getActiveRepository();
};

$scope.isLoadingLocation = function () {
  return $repositories.isLoadingLocation();
};

$scope.hasActiveLocation = function () {

```

```
return $repositories.hasActiveLocation();  
};
```

7.6.4 Repository setup

Create a repository, import the `airports.ttl` dataset, and execute the following SPARQL insert to add direct links for flights:

```
PREFIX onto: <http://www.ontotext.com/>  
INSERT {  
  ?node onto:hasFlightTo ?destination .  
} WHERE {  
  ?flight <http://openflights.org/resource/route/sourceId> ?node .  
  ?flight <http://openflights.org/resource/route/destinationId> ?destination .  
}
```

Now we will search for paths between airports based on the `hasFlightTo` predicate. Prior to this, remember to enable the Autocomplete index for your repository by following the steps described [here](#).

7.6.5 Select departure and destination airport

Now let's add inputs using Autocomplete to select the departure and destination airports. Inside the repository-is-set diff, add the two fields. Note the `visual-callback="findPath(startNode, uri)"` snippet that defines the callback to be executed once a value is selected through the Autocomplete. `uri` is the value from the Autocomplete. The following code sets the `starNode` variable in Angular and calls the `findPath` function when the destination is given. You can find out how to define this function in the scope a little further down in this tutorial.

```
<div class="card mb-2">  
  <div class="card-block">  
    <h3>From</h3>  
    <p>Search for a start node</p>  
    <search-resource-input class="search-rdf-resources"  
      namespacespromise="getNamespacesPromise"  
      autocompletepromisestatus="getAutocompletePromise"  
      text-button=""  
      visual-button="Show"  
      visual-callback="startNode = uri"  
      empty="empty"  
      preserve-input="true">  
    </search-resource-input>  
  </div>  
</div>  
<div class="card mb-2">  
  <div class="card-block">  
    <h3>To</h3>  
    <p>Search for an end node</p>  
    <search-resource-input class="search-rdf-resources"  
      namespacespromise="getNamespacesPromise"  
      autocompletepromisestatus="getAutocompletePromise"  
      text-button=""  
      visual-button="Show"  
      visual-callback="findPath(startNode, uri)"  
      empty="empty"  
      preserve-input="true">  
    </search-resource-input>  
  </div>  
</div>
```


They need the `getNamespacesPromise` and `getAutocompletePromise` to fetch the Autocomplete data. They should be initialized once the repository has been set in the controller.

```
function initForRepository() {
  if (!$repositories.getActiveRepository()) {
    return;
  }
  $scope.getNamespacesPromise = ClassInstanceDetailsService.getNamespaces($scope.
  ↪getActiveRepository());
  $scope.getAutocompletePromise = AutocompleteService.checkAutocompleteStatus();
}

$scope.$on('repositoryIsSet', function(event, args) {
  initForRepository();
});
initForRepository();
```

Note that both of these functions need to be called when the repository is changed, because you need to make sure that Autocomplete is enabled for this repository, and fetch the namespaces for it. Now you can autocomplete in your page.

Graph Paths ⓘ

From

Search for a start node

Sofia <http://openflights.org/resource/airport/id/1194>

To

Search for an end node

7.6.6 Find the paths between the selected airports

Now let's implement the `findPath` function in the scope. It finds all paths between nodes by using a simple Depth First Search algorithm. For each node, you can obtain its siblings with a call to the **rest/explore-graph/links** endpoint. This is the same endpoint the Visual graph is using to expand node links. Note that it is not part of the GraphDB API, but we will reuse it for simplicity. As an alternative, you can also obtain the direct links of a node by sending a SPARQL query to GraphDB.

Note: This is a demo implementation. For each repository containing a lot of links, the proposed approach is not appropriate, as it will send a request to the server for each node. This will quickly result in a huge amount of requests, which will very soon flood the browser.

```
var maxPathLength = 3;

var findPath = function (startNode, endNode, visited, path) {
  // A path is found, return a promise that resolves to it
  if (startNode === endNode) {
```

```

    return $q.when(path)
  }
  // Find only paths with maxLength, we want to cut only short paths between airports
  if (path.length === maxPathLength) {
    return $q.when([])
  }
  return $http({
    url: 'rest/explore-graph/links',
    method: 'GET',
    params: {
      iri: startNode,
      config: 'default',
      linksLimit: 50
    }
  }).then(function (response) {
    // Use only links with the hasFlightTo predicate
    var flights = _.filter(response.data, function(r) {return r.predicates[0] == "hasFlightTo"});
    // For each links, continue to search path recursively
    var promises = _.map(flights, function (link) {
      var o = link.target;
      if (!visited.includes(o)) {
        return findPath(o, endNode, visited.concat(o), path.concat(link));
      }
      return $q.when([]);
    });
    // Group together all promises that resolve to paths
    return $q.all(promises);
  }, function (response) {
    var msg = getError(response.data);
    toastr.error(msg, 'Error looking for path node');
  });
}

$scope.findPath = function (startNode, endNode) {
  findPath(startNode, endNode, [startNode], []).then(function (linksFound) {
    renderGraph(_.flattenDeep(linksFound));
  });
}

```

The findPath recursive function returns all the promises that will or will not resolve to paths. Each path is a collection of links. When all promises are resolved, you can flatten the array to obtain all links from all paths and draw one single graph with these links. Graph drawing is done with D3 in the renderGraph function. It needs a graph-visualization element to draw the graph inside. Add it inside the repository-is-set element below the autocomplete divs. Additionally, import graphs-vizualizations.css to reuse some styles.

```

<div class="card mb-2">
  ..
</div>
<div class="card mb-2">
  ...
</div>
<div class="graph-visualization"></div>

```

```
<link href="css/graphs-vizualizations.css" rel="stylesheet"/>
```

Now add the renderGraph render function mentioned above:

```

var width = 1000,
    height = 1000;

var nodeLabelRectScaleX = 1.75;

```

```

var force = d3.layout.force()
    .gravity(0.07)
    .size([width, height]);

var svg = d3.select(".main-container .graph-visualization").append("svg")
    .attr("viewBox", "0 0 " + width + " " + height)
    .attr("preserveAspectRatio", "xMidYMid meet");

function renderGraph(linksFound) {
    var graph = new Graph();

    var nodesFromLinks = _.union(_.flatten(_.map(linksFound, function (d) {
        return [d.source, d.target];
    })));

    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 + ")";
});

var predicate = container.selectAll(".link-wrapper")
.append("text")
.text(function (d, index) {
    return d.predicates[0];
})
.attr("class", function (d) {
    if (d.predicates.length > 1) {
        return "predicates";
    }
    return "predicate";
})
.attr("dy", "-0.5em")
.style("text-anchor", "middle")
.style("display", "")
.on("mouseover", function (d) {
    d3.event.stopPropagation();
});

var node = container.selectAll(".node")
.data(graph.nodes)
.enter().append("g")
.attr("class", "node-wrapper")
.attr("id", function (d) {
    return d.iri;
})
.append("circle")
.attr("class", "node")
.attr("r", function (d) {
    return d.size;
})
.style("fill", function (d) {
    return "rgb(255, 128, 128)";
})

var nodeLabels = container.selectAll(".node-wrapper").append("foreignObject")
.style("pointer-events", "none")
.attr("width", function (d) {
    return d.size * 2 * nodeLabelRectScaleX;
});
// height will be computed by updateNodeLabels

updateNodeLabels(nodeLabels);

function updateNodeLabels(nodeLabels) {
    nodeLabels.each(function (d) {
        d.fontSize = D3.Text.calcFontSizeRow(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/messes up other unrelated elements. div seems to work too.
    .append("xhtml:div")
    .attr("class", "node-label-body")
    .style("font-size", function (d) {
        return d.fontSize + 'px';
    })
    .append('xhtml:div')
    .text(function (d) {
        return d.labels[0].label;
    });
}

// Update positions on tick
force.on("tick", function () {

    // recalculate links attributes
    link.attr("x1", function (d) {
        return d.source.x;
    }).attr("y1", function (d) {
        return d.source.y;
    }).attr("x2", function (d) {
        return d.target.x;
    }).attr("y2", function (d) {
        return d.target.y;
    });

    // recalculate predicates attributes
    predicate.attr("x", function (d) {
        return d.x = (d.source.x + d.target.x) * 0.5;
    }).attr("y", function (d) {
        return d.y = (d.source.y + d.target.y) * 0.5;
    });

    // recalculate nodes attributes
    node.attr("cx", function (d) {
        return d.x;
    }).attr("cy", function (d) {
        return d.y;
    });

    nodeLabels.attr("x", function (d) {
        return d.x - (d.size * nodeLabelRectScaleX);
    }).attr("y", function (d) {
        // the height of the nodeLabel box is 3 times the fontSize computed by updateNodeLabels
        // and we want to offset it so that its middle matches the centre of the circle, hence_
↪divided by 2
        return d.y - 3 * d.fontSize / 2;
    });

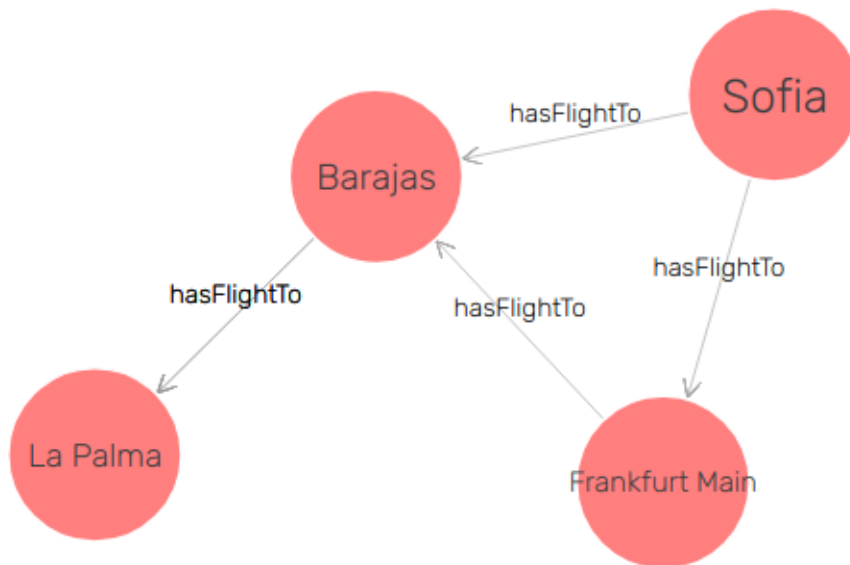
});
force.start();
}

```

It obtains the URIs for the nodes from all links, and finds their labels through calls to the **rest/explore-graph/node** endpoint. A graph object is defined to represent the visual abstraction, which is simply a collection of nodes and links. The `draw(graph)` function does the D3 drawing itself using the D3 force layout.

7.6.7 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 request to the server. The purpose of this tutorial is to introduce you to the Workbench extension with a naive paths prototype.

7.6.8 Add status message

Noticing that path finding can take some time, we may want to add a message for the user.

```

$scope.findPath = function (startNode, endNode) {
  $scope.pathFinding = true;
  findPath(startNode, endNode, [startNode], []).then(function (linksFound) {
    $scope.pathFinding = false;
    renderGraph(_.flattenDeep(linksFound));
  });
}

```

```

<div ng-show="pathFinding">Looking for all paths between nodes...</div>
<div class="graph-visualization"></div>

```

The source code for this example can be found in the [workbench-paths-example](#) GitHub project.

7.7 Workbench REST API

7.7.1 Repository management with the Workbench REST API

The GraphDB Workbench 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 Workbench REST API.

7.7.1.1 Prerequisites

- One or optionally two machines with Java.
- One GraphDB instance:
 - Start GraphDB Free, SE or EE on the first machine.

Tip: For more information on deploying GraphDB, please see one of:

- * Installing GraphDB Free
 - * Installing GraphDB SE
 - * Installing GraphDB EE
-

- Another GraphDB instance (optional, needed for the *attaching a remote location example*):
 - Start GraphDB Free, SE or EE 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/`, for the first instance;
- `http://192.0.2.2:7200/`, for the second instance.

Please adjust the URLs according to the IPs or hostnames of your own machines.

7.7.1.2 Managing repositories

Create a repository

Repositories can be created by providing a TTL file with all the configuration parameters.

First, download the sample repository config file `repo-config.ttl`.

Then, send the file with a POST request using the following `curl` command:

```
curl -X POST\
  http://192.0.2.1:7200/rest/repositories\
  -H 'Content-Type: multipart/form-data'\
  -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:FreeSailRepository",
    "location": "",
    "readable": true,
    "writable": true,
    "local": true
  }
]
```

7.7.1.3 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.

Activate a location

Use the following `curl` command to activate the previously attached location by sending a POST request to the API:

- Activate the location `http://192.0.2.2:7200/` on `http://192.0.2.1:7200/`:

```
curl -X POST http://192.0.2.1:7200/rest/locations/activate\
-H 'Content-Type:application/json'\
-d '{
  "uri": "http://192.0.2.2:7200/"
}'
```

Note: The default GraphDB location (stored locally on disk) is already activated by default. Activating an already active location is not an error. Note also that activating a location is not required for managing it through the REST API as an explicit location can be provided as a parameter to all REST API calls.

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 1 local location and 1 remote location:

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

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/`:

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

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

7.7.1.4 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 Documentation* menu.

7.7.2 Cluster management with the Workbench REST API

The GraphDB Workbench REST API can be used for managing a GraphDB EE cluster. It includes connecting workers to masters, connecting masters to each other as well monitoring the state of a cluster. This tutorial shows

how to use `curl` command to perform basic cluster management through the Workbench REST API.

This tutorial builds upon the tutorial *Repository management with the Workbench REST API*.

7.7.2.1 Prerequisites

- Four machines with Java and Tomcat.
- Three GraphDB EE Server instances to host one master and two worker repositories:
Start GraphDB EE on all machines.

Tip: See Installing GraphDB EE for more information on installing GraphDB and setting up JMX.

- One GraphDB EE Workbench instance to serve as a REST API endpoint:
Start GraphDB EE (this can be done on your local machine too).
- `curl` command line tool for sending requests to the API.

Hint: Throughout the tutorial, the four instances will be referred to with the following URLs:

- `http://192.0.2.1:7200/`, for the instance that will host the first worker;
- `http://192.0.2.2:7200/`, for the instance that will host the second worker;
- `http://192.0.2.3:7200/`, for the instance that will host the master;
- `http://192.0.2.4:7200/`, for the Workbench instance.

Please adjust the URLs according to the IPs or hostnames of your own machines.

7.7.2.2 Creating a cluster

Attach the locations

To create a remote location, send a POST request to the REST API at `http://192.0.2.4:7200/`:

1. Attach the location `http://192.0.2.1:7200/`

```
curl -X PUT http://192.0.2.4:7200/rest/locations\  
-H 'Content-Type:application/json\  
-d '{  
  "uri": "http://192.0.2.1:7200/"  
}'
```

2. Attach the location `http://192.0.2.2:7200/`

```
curl -X PUT http://192.0.2.4:7200/rest/locations\  
-H 'Content-Type:application/json\  
-d '{  
  "uri": "http://192.0.2.2:7200/"  
}'
```

3. Attach the location `http://192.0.2.3:7200/`:

```
curl -X PUT http://192.0.2.4:7200/rest/locations\  
-H 'Content-Type:application/json\  
-d '{  
  "uri": "http://192.0.2.3:7200/"  
}'
```

Inspect the result in the GraphDB Workbench by accessing the *Setup -> Repositories* view:

Locations and Repositories

Repository ID	Description	Actions	Permissions
SYSTEM	System configuration repository		

Create repository

Inactive locations

Inactive	http://192.0.2.2:8080/graphdb-server	
Inactive	http://192.0.2.3:8080/graphdb-server	

Attach Location

Note: Note that the first attached location has become active but that is irrelevant for this tutorial.

Create workers and master

After successfully attaching the three remote locations you have to create the repositories.

First, download the sample repository config files `master1-config.ttl`, `worker1-config.ttl` and `worker2-config.ttl`.

Then, create each repository by sending a PUT request to the REST API at `http://192.0.2.4:7200/`:

1. Create worker1 repository on `http://192.0.2.1:7200/`:

```
curl -X POST\
  http://192.0.2.4:7200/rest/repositories?location=http://192.0.2.1:7200/\
  -H 'Content-Type: multipart/form-data'\
  -F "config=@worker1-config.ttl"
```

2. Create worker2 repository on `http://192.0.2.2:7200/`:

```
curl -X POST\
  http://192.0.2.4:7200/rest/repositories?location=http://192.0.2.2:7200/\
  -H 'Content-Type: multipart/form-data'\
  -F "config=@worker2-config.ttl"
```

3. Create master1 repository on `http://192.0.2.3:7200/`:

```
curl -X POST\
  http://192.0.2.4:7200/rest/repositories?location=http://192.0.2.3:7200/\
  -H 'Content-Type: multipart/form-data'\
  -F "config=@master1-config.ttl"
```

Connect workers to master

To connect the workers to the master, send POST requests for each worker:

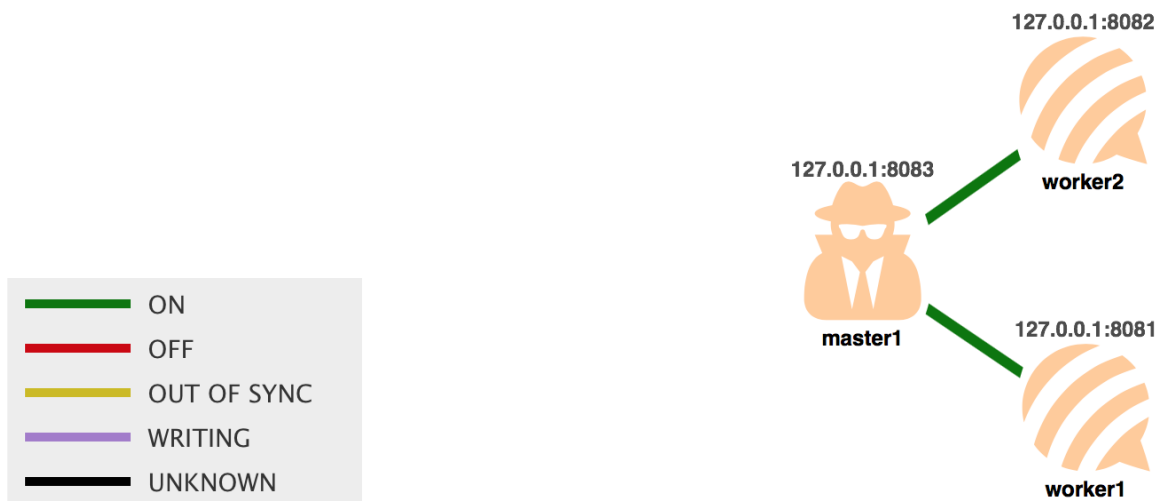
1. Connect worker1 with master1

```
curl -X POST http://192.0.2.4:7200/rest/cluster/masters/master1/workers\
-H 'Content-Type: application/json'\
-d '{
  "workerURL": "http://192.0.2.1:7200/repositories/worker1",
  "masterLocation": "http://192.0.2.3:7200/"
}'
```

2. Connect worker2 with master1

```
curl -X POST http://192.0.2.4:7200/rest/cluster/masters/master1/workers\
-H 'Content-Type: application/json'\
-d '{
  "workerURL": "http://192.0.2.2:7200/repositories/worker2",
  "masterLocation": "http://192.0.2.3:7200/"
}'
```

After the successful execution of these requests you should be able to see the following in the *Setup -> Cluster* view in the GraphDB Workbench:



7.7.2.3 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 Documentation* menu.

7.7.3 Workbench REST API curl commands

This page displays GraphDB Workbench REST API calls as curl commands which enables developers to script these calls in their applications.

This data is also available at <http://localhost:7200/webapi> on a GraphDB instance running on the default 7200 port.

In addition to this, the [RDF4J API](#) is also accessible.

7.7.3.1 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.
- **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 /rest/locations
Get all connected GraphDB locations
```

```
curl <base_url>/rest/locations -H 'Accept: application/json'
```

```
GET /rest/locations/active
Get active connected GraphDB location
```

```
curl <base_url>/rest/locations/active -H 'Accept: application/json'
```

```
POST /rest/locations
Modify settings for a connected GraphDB location
```

```
curl -X POST <base_url>/rest/locations -H 'Content-Type:application/json' -H 'Accept:text/plain' -d '
{
  "username": "<username>",
  "password": "<password>",
  "uri": "<location_uri>"
}'
```

```
PUT /rest/locations
Connect a local or remote GraphDB location
```

```
curl -X PUT <base_url>/rest/locations -H 'Content-Type:application/json' -H 'Accept:text/plain' -d '
{
  "username": "<username>",
  "password": "<password>",
  "uri": "<location_uri>"
}'
```

```
POST /rest/locations/activate
Activate a connected GraphDB location
```

```
curl -X POST <base_url>/rest/locations/activate -H 'Content-Type:application/json' -H 'Accept:text/
↩plain' -d '
{
  "username": "<username>",
```

```
"password": "<password>",
"uri": "<location_uri>"
}'
```

```
DELETE /rest/locations
Disconnect a GraphDB location
```

```
curl -X DELETE <base_url>/rest/locations?uri=<encoded_location_uri> -H 'Accept: text/plain'
```

```
POST /rest/locations/default-repository
Set the default repository
```

```
curl -X POST <base_url>/rest/locations/default-repository -H 'Content-Type:application/json' -d '
{
  "repository": "<repo_id>"
}'
```

```
GET /rest/locations/id
Get the location ID
```

```
curl <base_url>/rest/locations/id
```

Hint: Common parameters:

<base_url> - the url host and path leading to the deployed GDB-Workbench webapp;

<location_uri> - file system path of the physical location of the repo (could be local or remote);

<encoded_location_uri> - url encoded file system path of the physical location of the repo (could be local or remote).

7.7.3.2 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.
- `sesameType` (string): The sesame type of the repository.
- `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 /rest/repositories
Get all repositories in the active location or another location
```

```
curl <base_url>/rest/repositories -H 'Accept: application/json'
```

```
GET /rest/repositories/defaultConfig/<repositoryType>
```

Get the default repository configuration **for** a repository **type**: "se", "master", "worker", "free"

```
curl <base_url>/rest/repositories/defaultConfig/<repositoryType> -H 'Accept: application/json'
```

```
GET /rest/repositories/{repositoryId}
```

Get repository configuration **in** the active location **or** another attached location

```
curl <base_url>/rest/repositories/<repo_id>?location=<encoded_location_uri> -H 'Accept: application/↵json'
```

```
GET /rest/repositories/cluster
```

Get **all** repositories of **all** nodes of a cluster

```
curl <base_url>/rest/repositories/cluster?location=<encoded_location_uri> -H 'Accept: application/↵json'
```

```
GET /rest/repositories/{repositoryId}/size
```

Get repository size

```
curl <base_url>/rest/repositories/<repo_id>/size?location=<encoded_location_uri> -H 'Accept: ↵application/json'
```

```
POST /rest/repositories
```

Create a repository **in** an attached GraphDB location (ttl file)

```
curl -X POST <base_url>/rest/repositories?location=<encoded_location_uri> -H 'Accept: application/↵json' -H 'Content-Type: multipart/form-data' -F "config=@<repo_ttl_config_filename>"
```

```
PUT /rest/repositories
```

Create a repository **in** an attached GraphDB location (params)

```
curl -X POST <base_url>/rest/repositories -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": "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>
}'

```

```
PUT /rest/repositories/{repositoryId}
Edit repository configuration
```

```
curl -X PUT <base_url>/rest/repositories/<repo_id> -H 'Accept: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": "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.

GET /rest/repositories/{repositoryId}/download
Download repository configuration **as** a Turtle file. Optionally provide a location parameter.

```
curl <base_url>/rest/repositories/<repo_id>/download?location=<encoded_location_uri>
```

DELETE /rest/repositories/{repositoryId}
Delete a repository **in** an attached GraphDB location

```
curl -X DELETE <base_url>/rest/repositories/<repo_id>?location=<encoded_location_uri> -H 'Accept:
↪application/json'
```

Hint: Common parameters:

<base_url> - the url host and path leading to the deployed GDB-Workbench webapp;

<location_uri> - file system path of the physical location of the repo (could be local or remote);

<encoded_location_uri> - url encoded file system path of the physical location of the repo (could be local or remote);

<repo_id> - the id string with which the current repository can be referred to;

<repo_title> - human-readable name of the current repository;

<repo_type> - type of the repository, could be *se*, *worker*, *master*.

7.7.3.3 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 recognised.
 - * failOnUnknownLanguageTags (boolean) - fail parsing if languages are not recognised.
 - * normalizeDataTypeValues (boolean) - normalize recognized datatypes values.
 - * normalizeLanguageTags (boolean) - normalize recognised 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 recognised datatypes.
 - * verifyLanguageTags (boolean) - verify language based on a given set of definitions for valid languages.

```
DELETE /rest/data/import/upload/{repositoryId}
Cancel import operation
```

```
curl -x DELETE <base_url>/rest/data/import/upload/<repo_id>?name=<encoded_data_url>
```

```
DELETE /rest/data/import/server/{repositoryId}
Cancel server files import operation
```

```
curl -X DELETE <base_url>/rest/data/import/server/<repo_id>?name=<encoded_filepath>
```

Hint: <encoded_filepath> - encoded filepath leading to a server file that is in the process of being imported.

```
GET /rest/data/import/upload/{repositoryId}
Get data urls for import
```

```
curl <base_url>/rest/data/import/upload/<repo_id>
```

```
GET /rest/data/import/server/{repositoryId}
Get server files available for import
```

```
curl <base_url>/rest/data/import/server/<repo_id>
```

```
POST /rest/data/import/upload/{repositoryId}/url
Import from data url into the repository
```

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '
{
  "baseURI": "string",
  "context": "string",
  "data": "string",
  "forceSerial": true,
  "format": "string",
  "message": "string",
  "name": "string",
  "parserSettings": {
    "failOnUnknownDataTypes": boolean,
    "failOnUnknownLanguageTags": boolean,
```

```

    "normalizeDataTypeValues": boolean,
    "normalizeLanguageTags": boolean,
    "preserveBNodeIds": boolean,
    "stopOnError": boolean,
    "verifyDataTypeValues": boolean,
    "verifyLanguageTags": boolean
  },
  "replaceGraphs": [
    "string"
  ],
  "status": "PENDING",
  "timestamp": 0,
  "type": "string"
}'
'<base_url>/rest/data/import/upload/<repo_id>/url'

```

POST /rest/data/import/server/{repositoryId}
 Import **from server** files into the repository

```

curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{
  "fileNames": [
    "<data_url>",
    "<data_url>"
  ]
}' '<base_url>/rest/data/import/server/<repo_id>'

```

Hint: Common parameters:

<base_url> - the url host and path leading to the deployed GDB-Workbench webapp;

<repo_id> - the id string with which the current repository can be referred to;

<encoded_repository_filepath> - encoded filepath leading to physical repo location;

<data_url> - url leading to the data source to be imported;

<encoded_data_url> - encoded url leading to the data source to be imported;

<encoded_server_filepath> - encoded filepath leading to server files to be imported.

7.7.3.4 Saved Queries

GET /rest/sparql/saved-queries
 Get saved query (**or** queries, **if** no parameter specified)

```
curl <base_url>/rest/sparql/saved-queries?name=<query_name>
```

POST /rest/sparql/saved-queries
 Create a new saved query

```

curl -X POST <base_url>/rest/sparql/saved-queries -d '{
  {
    "body": "<query_body>",
    "name": "<query_name>"
  }
}'

```

PUT /rest/sparql/saved-queries
 Edit an existing saved query

```
curl -X PUT <base_url>/rest/sparql/saved-queries -d '{
  "body": "<query_body>",
  "name": "<query_name>"
}'
```

DELETE /rest/sparql/saved-queries
Delete an existing saved query

```
curl -X DELETE <base_url>/rest/sparql/saved-queries?name=<query_name>
```

7.7.3.5 Cluster Management

GET /rest/cluster/masters/{masterRepositoryId}
Get information about master

```
curl <base_url>/rest/cluster/masters/<master_repo_id>
```

POST /rest/cluster/masters/{masterRepositoryId}
Set master attribute

```
curl -X POST <base_url>/rest/cluster/masters/<master_repo_id>?masterLocation=<master_location> --
  header 'Content-Type: application/json' --header 'Accept: text/plain' -d '<inline_bean_model>'
```

GET /rest/cluster/masters/{masterRepositoryId}/workers
Get workers connected to a master

```
curl <base_url>/rest/cluster/masters/<master_repo_id>/workers?masterLocation=<master_location>
```

DELETE /rest/cluster/masters/{masterRepositoryId}/workers
Disconnect a worker **from** a master

```
curl -X DELETE <base_url>/rest/cluster/masters/<master_repo_id>/workers?masterLocation=<master_
  location>\&workerURL=<worker_url>
```

DELETE /rest/cluster/masters/{masterRepositoryId}/peers
Disconnect two masters

```
curl -X DELETE <base_url>/rest/cluster/masters/<master_repo_id>/peers?masterLocation=<master_
  location>\&masterNodeID=<master_node_id>\&peerLocation=<peer_location>\&peerNodeID=<peer_node_id>
  \&peerRepositoryID=<peer_repo_id>
```

POST /rest/cluster/masters/{masterRepositoryId}/peers
Connect two masters

```
curl -X POST <base_url>/rest/cluster/masters/<master_id>/peers -H 'Accept: application/json' -H
  'Content-Type: application/json' -d '{
  {
    "bidirectional": boolean,
    "masterLocation": "<master_location>",
    "masterNodeID": "<master_node_id>",
    "peerLocation": "<peer_location>",
    "peerNodeID": "<peer_node_id>",
    "peerRepositoryID": "<peer_repo_id>"
  }
}'
```


POST /rest/cluster/masters/{masterRepositoryId}/workers
Connect a worker to a master

```
curl -X POST <base_url>/rest/cluster/masters/<master_repo_id>/workers -H 'Accept: application/json' -H 'Content-Type: application/json' -d '{
  "workerURL": "<worker_url>",
  "masterLocation": "<master_location>"
}'
```

POST /rest/cluster/nodes/clone
Clone a worker

```
curl -X POST <base_url>/rest/cluster/nodes/clone -H 'Accept: application/json' -H 'Content-Type: application/json' -d '{
  "cloningNodeLocation": "<clone_node_location>",
  "cloningNodeRepositoryID": "<clone_node_repo_id>",
  "newNodeLocation": "<new_node_location>",
  "newNodeRepositoryID": "<new_node_repo_id>",
  "newNodeTitle": "<new_node_repo_title>"
}'
```

Hint: <clone_node_location> - current physical location of the worker repository to be cloned;
 <clone_node_repo_id> - the string id of the current physical location of the worker repository to be cloned;
 <new_node_location> - new physical location of the new *worker* repository to be created;
 <new_node_repo_id> - the string id of the new physical location of the new worker repository to be created;
 <new_node_repo_title> - new title of the new *worker* repository to be created.

GET /rest/cluster/masters/{masterRepositoryId}/backup
Initiate a cluster backup

```
curl <base_url>/rest/cluster/masters/<master_repo_id>/backup?masterLocation=<master_location>&
masterRepositoryId=<master_repo_id>&backupName=<backup_name>
```

GET /rest/cluster/masters/{masterRepositoryId}/restore
Initiate a cluster restore

```
curl <base_url>/rest/cluster/masters/<master_repo_id>/restore?masterLocation=<master_location>&
masterRepositoryId=<master_repo_id>&backupName=<backup_name>
```

Hint: Common parameters:

<base_url> - the url host and path leading to the deployed GDB-Workbench webapp;
 <master_repo_id> - the id string with which the current repository of type *master* can be referred to;
 <master_location> - file path leading to physical master repository location. Use *local* as value when referred from and to the same location;
 <worker_url> - absolute url referring to a worker repository;
 <backup_name> - the id string referring to a new backup of a repository or to an old one when attempting to restore it.

7.8 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.

7.8.1 Related to google people and organization in factforge.net

The following example fetches people and organizations related to Google. One of the sample queries in factforge.net is used and rewritten into a construct query. Type is used to differ entities of different types.

```
<html>
<body>
<!-- Include the library -->
<script src="../../lib/ogma.min.js"></script>
<script src="../../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../../lib/lodash.js"></script>
<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole_
screen. -->
<div id="graph-container" style="position: absolute; left: 0; top: 0; bottom: 0; right: 0;"></div>

<script>
// Which namespace to chose types from
var dboNamespace = "http://dbpedia.org/ontology"

// One of factforge saved queries enriched with types and rdf rank
var peopleAndOrganizationsRelatedToGoogle = `
    # F03: People and organizations related to Google
    # - picks up people related through any type of relationships
    # - picks up parent and child organizations
    # - benefits from inference over transitive dbo:parent
    # - RDRank makes it easy to see the "top suspects" in a list of 94 entities
    # Change Google with any organization, e.g. type dbr:Hew and Ctrl-Space to auto-complete

    PREFIX dbo: <http://dbpedia.org/ontology/>
    PREFIX rank: <http://www.ontotext.com/owlim/RDRank#>
    PREFIX dbr: <http://dbpedia.org/resource/>
    PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
    CONSTRUCT {
        dbr:Google ?p2 ?related_entity .
        dbr:Google sesame:directType ?type .
        ?related_entity ?p1 dbr:Google .
        ?related_entity sesame:directType ?entity_type .
        ?related_entity rank:hasRDRank ?related_entity_rank .
        dbr:Google dbr:hasChildOrParentOrg ?related_organization .
        ?related_organization sesame:directType ?org_type .
        ?related_organization rank:hasRDRank ?related_org_rank .
    }
    WHERE {
        BIND( dbr:Google AS ?entity )
        {
            ?related_entity a dbo:Person; ?p1 ?entity .
            FILTER(?p1 NOT IN (dbo:wikiPageWikiLink)) .
            ?related_entity sesame:directType ?entity_type .
            ?related_entity rank:hasRDRank ?related_entity_rank .
        }
        UNION
        {
```

```

        ?related_entity a dbo:Person .
        ?entity ?p2 ?related_entity .
        FILTER(?p2 NOT IN (dbo:wikiPageWikiLink)) .
        ?related_entity sesame:directType ?entity_type .
        ?related_entity rank:hasRDFRank ?related_entity_rank .
    }
    UNION
    {
        ?related_organization a dbo:Organisation ; (dbo:parent | ^dbo:parent) ?entity .
        ?related_organization sesame:directType ?org_type .
        ?related_organization rank:hasRDFRank ?related_org_rank .
    } UNION {
        dbr:Google sesame:directType ?type .
    }
}
\

var postData = {
  query: peopleAndOrganizationsRelatedToGoogle,
  infer: true,
  sameAs: true,
  limit: 1000,
  offset: 0
}

$.ajax({
  url: graphDBRepoLocation,
  type: 'POST',
  data: postData,
  headers: {
    'Accept': 'application/rdf+json'
  },
  success: function (data) {

    // Converts rdf+json to a simple list of triples
    var triples = convertData(data);

    // Get all nodes uris
    var linkTriples = _.filter(triples, function (triple) {
      return triple[1] !== rankPredicate && triple[1] !== typePredicate
    });
    var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
      return t[0]
    }), _.map(linkTriples, function (t) {
      return t[2]
    })));

    // Get triples for rdf rank
    var ranks = _.filter(triples, function (triple) {
      return triple[1] === rankPredicate
    });

    // Get triples for types
    var typeTriples = _.filter(triples, function (triple) {
      return triple[1] === typePredicate && triple[2].indexOf(dboNamespace) === 0
    });

    // Create node objects
    var nodes = _.map(nodesUris, function (nUri) {
      var rank = _.find(ranks, function (rankTriple) {
        return rankTriple[0] === nUri && rankTriple[1] === rankPredicate
      });
    });
  }
});

```

```

        var type = _.find(typeTriples, function (typeTriple) {
            return typeTriple[0] === nUri && typeTriple[1] === typePredicate
        });
        return {
            id: nUri,
            text: getLocalName(nUri) + (type !== undefined ? " (" +
getLocalName(type[2]) + ")" : ""),
            size: ((rank !== undefined) ? rank[2] * 100 : 5),
            color: ((type !== undefined) ? stringToColour(type[2]) : "#eceeef"),
        }
    });

    // Create edge objects
    var edges = _.map(linkTriples, function (triple, index) {
        return {
            id: index,
            source: triple[0],
            target: triple[2],
            text: getLocalName(triple[1]),
            shape: 'arrow',
            size: 0.2
        }
    });

    // Initialize ogma with the data
    var ogma = new Ogma({
        container: 'graph-container',
        settings: {
            texts: {
                nodeFontSize: 20,
                edgeFontSize: 15,
                nodeSizeThreshold: 0
            }
        },
        graph: {
            nodes: nodes,
            edges: edges
        }
    });

    ogma.locate.center();

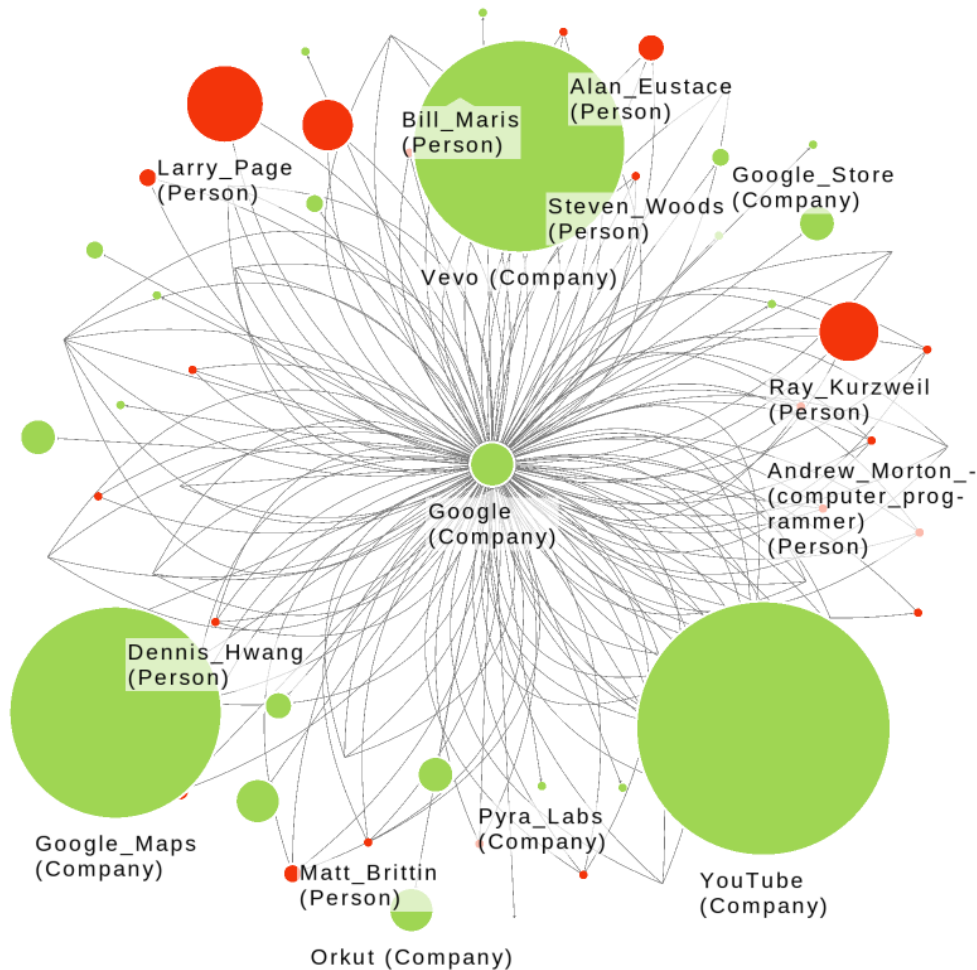
    ogma.layouts.start('forceLink', {}, {
        // sync parameters
        onEnd: endLayout
    });

    function endLayout() {
        ogma.locate.center({
            easing: 'linear',
            duration: 300
        });
    }
}

})
</script>
</body>
</html>

```

Which produces the following graph:



7.8.2 Suspicious control chain through off-shore company in factforge.net

The following example fetches suspicious control chain through off-shore company, which is another saved query in factforge.net rewritten as a graph query. Except for the entities, their RDF Rank and their type is fetched. Nodes size is based on RDF Rank and node color on its type. All of the examples use a commons.js file with some common function, i.e data model conversion.

```
<html>
<body>
<!-- Include the library -->
<script src="../../lib/ogma.min.js"></script>
<script src="../../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../../lib/lodash.js"></script>

<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole_
screen. -->
<div id="graph-container" style="position: absolute; left: 0; top: 0; bottom: 0; right: 0;"></div>

<script>

// Which namespace to chose types from
var dboNamespace = "http://dbpedia.org/ontology"

var suspiciousOffshore = `
```

```
# F05: Suspicious control chain through off-shore company

PREFIX onto: <http://www.ontotext.com/>
PREFIX fibo-fnd-rel-rel: <http://www.omg.org/spec/EDMC-FIBO/FND/Relations/Relations/>
PREFIX ff-map: <http://factforge.net/ff2016-mapping/>
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>
PREFIX dbo: <http://dbpedia.org/ontology/>

CONSTRUCT {
  ?c1 fibo-fnd-rel-rel:controls ?c2 .
  ?c2 fibo-fnd-rel-rel:controls ?c3 .
  ?c1 ff-map:primaryCountry ?c1_country .
  ?c2 ff-map:primaryCountry ?c2_country .
  ?c3 ff-map:primaryCountry ?c3_country .
  ?c1 sesame:directType ?t1 .
  ?c2 sesame:directType ?t2 .
  ?c3 sesame:directType ?t3 .
  ?c1_country sesame:directType dbo:Country .
  ?c3_country sesame:directType dbo:Country .
  ?c3_country sesame:directType dbo:Country .
} FROM onto:disable-sameAs
WHERE {
  ?c1 fibo-fnd-rel-rel:controls ?c2 .
  ?c2 fibo-fnd-rel-rel:controls ?c3 .
  ?c1 sesame:directType ?t1 .
  ?c2 sesame:directType ?t2 .
  ?c3 sesame:directType ?t3 .
  ?c1 ff-map:primaryCountry ?c1_country .
  ?c2 ff-map:primaryCountry ?c2_country .
  ?c3 ff-map:primaryCountry ?c1_country .
  FILTER (?c1_country != ?c2_country)

  ?c2_country ff-map:hasOffshoreProvisions true .
} `

var postData = {
  query: suspiciousOffshore,
  infer: true,
  sameAs: true,
  limit: 1000,
  offset: 0
}

$.ajax({
  url: graphDBRepoLocation,
  type: 'POST',
  data: postData,
  headers: {
    'Accept': 'application/rdf+json'
  },
  success: function (data) {

    var triples = convertData(data);

    // Get all nodes uris
    var linkTriples = _.filter(triples, function (triple) {
      return triple[1] !== typePredicate
    });
    var nodesUris = _.uniq(_.union(_.map(linkTriples, function (t) {
      return t[0]
    })), _.map(linkTriples, function (t) {
```

```

        return t[2]
    }));

    // Get triples for types
    var typeTriples = _.filter(triples, function (triple) {
        return triple[1] === typePredicate && triple[2].indexOf(dboNamespace) === 0
    });

    // Create node objects
    var nodes = _.map(nodesUris, function (nUri) {
        var type = _.find(typeTriples, function (typeTriple) {
            return typeTriple[0] === nUri && typeTriple[1] === typePredicate
        });
        return {
            id: nUri,
            text: getLocalName(nUri) + (type !== undefined ? " (" +
getLocalName(type[2]) + ")" : ""),
            size: 5,
            color: ((type !== undefined) ? stringToColour(type[2]) : "#eceeef"),
        }
    });

    // Create edge objects
    var edges = _.map(linkTriples, function (triple, index) {
        return {
            id: index,
            source: triple[0],
            target: triple[2],
            text: getLocalName(triple[1]),
            shape: 'arrow',
            size: 0.5
        }
    });

    // Initialize ogma with the data
    var ogma = new Ogma({
        container: 'graph-container',
        settings: {
            texts: {
                nodeFontSize: 20,
                edgeFontSize: 15,
                nodeSizeThreshold: 0,
                edgeSizeThreshold: 0
            }
        },
        graph: {
            nodes: nodes,
            edges: edges
        }
    });

    ogma.locate.center();

    ogma.layouts.start('forceLink', {}, {
        onEnd: endLayout
    });

    function endLayout() {
        ogma.locate.center({
            easing: 'linear',
            duration: 300
        });
    }
}

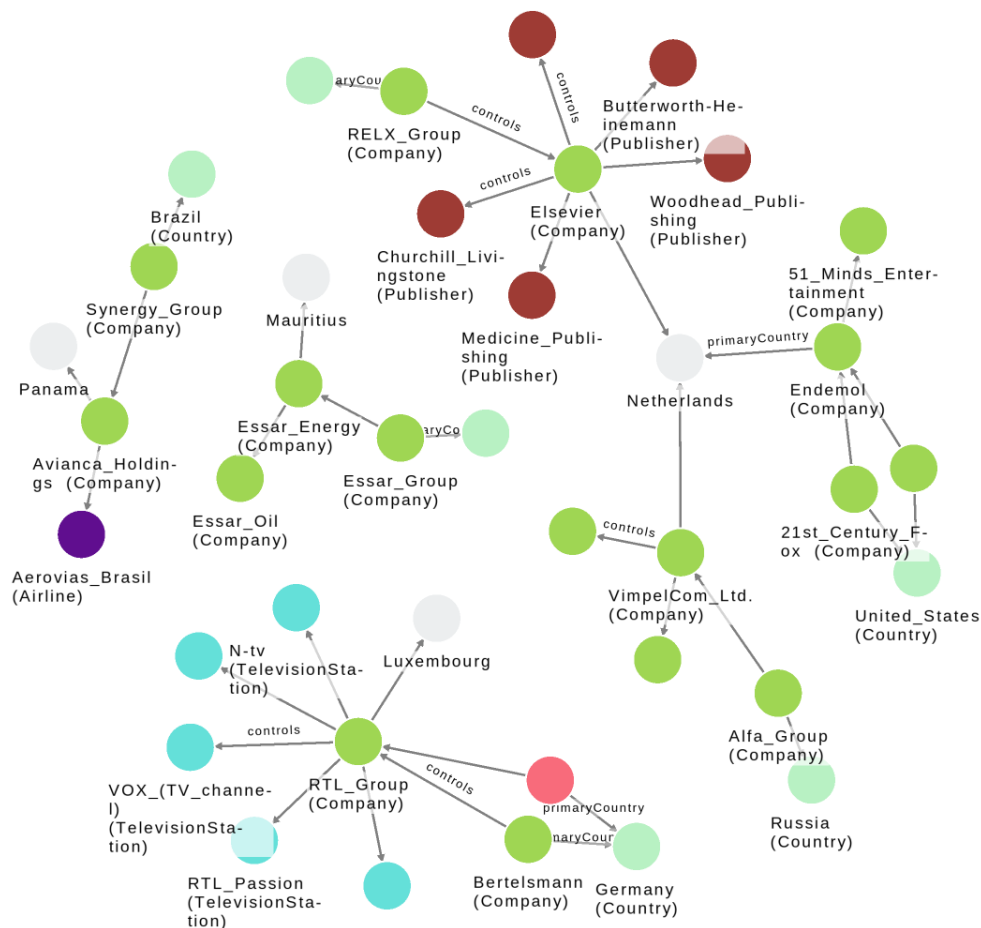
```

```

    }
  })
</script>
</body>
</html>

```

Which produces the following graph:



7.8.3 Flights shortest path

Import the `airports.ttl` dataset which contains airports and flights. Display the airports on a map using the latitude and longitude properties. Find the shortest path between airports in terms of number of flights.

```

<html>
<body>
<!-- Include the library -->
<script src="../lib/ogma.min.js"></script>
<script src="../lib/jquery-3.2.0.min.js"></script>
<script src="commons.js"></script>
<script src="../lib/lodash.js"></script>

<style>
  #graph-container { top: 0; bottom: 0; left: 0; right: 0; position: absolute; margin: 0;
  overflow: hidden; }
  .info {
    position: absolute;
    color: #fff;
    background: #141229;

```



```

    font-size: 12px;
    font-family: monospace;
    padding: 5px;
  }
  .info.n { top: 0; left: 0; }
</style>

<!-- This div is the DOM element containing the graph. The style ensures that it takes the whole
↪screen. -->
<div id="graph-container"></div>
<div id="n" class="info n">loading a large graph, it can take a few seconds...</div>

<script>

// The query to visualize
var airportsQuery = `
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
construct {
    ?source <http://openflights.org/resource/route/hasFlightTo> ?dest.
    ?dest rdf:type ?dtype .
    ?dest rdfs:label ?destLabel .
    ?source rdf:type ?stype .
    ?source rdfs:label ?sLabel .
    ?source <http://openflights.org/resource/airport/latitude> ?sourceLat .
    ?dest <http://openflights.org/resource/airport/latitude> ?destLat .
    ?source <http://openflights.org/resource/airport/longitude> ?sourceLong .
    ?dest <http://openflights.org/resource/airport/longitude> ?destLong .
} where {
    ?flight <http://openflights.org/resource/route/destinationId> ?dest .
    ?flight <http://openflights.org/resource/route/sourceId> ?source .
    ?flight rdf:type ?ftype .
    ?dest rdf:type ?dtype .
    ?dest rdfs:label ?destLabel .
    ?source rdf:type ?stype .
    ?source rdfs:label ?sLabel .
    ?source <http://openflights.org/resource/airport/latitude> ?sourceLat .
    ?dest <http://openflights.org/resource/airport/latitude> ?destLat .
    ?source <http://openflights.org/resource/airport/longitude> ?sourceLong .
    ?dest <http://openflights.org/resource/airport/longitude> ?destLong .
}
`;

var typePredicate = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type";
var labelPredicate = "http://www.w3.org/2000/01/rdf-schema#label";
var latitudePredicate = "http://openflights.org/resource/airport/latitude";
var longitudePredicate = "http://openflights.org/resource/airport/longitude";

var postData = {
  query: airportsQuery,
  infer: true,
  sameAs: true,
  // limit: 10000
}

var startNode = 'http://openflights.org/resource/airport/id/1194';
var endNode = 'http://openflights.org/resource/airport/id/4061';

$.ajax({
  url: 'http://localhost:8082/repositories/airroutes',
  type: 'POST',

```

```

data: postData,
headers: {
    'Accept': 'application/rdf+json'
},
success: function (data) {

    var triples = convertData(data);

    // Get all nodes uris
    var linkTriples = _.filter(triples, function (triple) {
        return triple[1] !== typePredicate && triple[1] !== labelPredicate &&
↪triple[1] !== latitudePredicate && triple[1] !== longitudePredicate
    });
    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 longitudeTriples = _.filter(triples, function (triple) {
        return triple[1] === longitudePredicate
    });

    // 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 longitude = _.find(longitudeTriples, function (longTriple) {
            return longTriple[0] === nUri && longTriple[1] ===
↪longitudePredicate
        });
        return {
            id: nUri,
            text: (label !== undefined) ? (label[2] + "(" + getLocalName(nUri) +
↪")" : getLocalName(nUri),
            size: 0.5,
            color: ((type !== undefined) ? stringToColour(type[2]) : "#eceeef"),
            latitude: (latitude !== undefined) ? parseFloat(latitude[2]) : 0,
            longitude: (longitude !== undefined) ? parseFloat(longitude[2]) : 0,
        }
    });

    // Create edge objects
    var edges = _.map(linkTriples, function (triple, index) {
        return {
            id: index,

```

```

        source: triple[0],
        target: triple[2],
        text: getLocalName(triple[1]),
        shape: 'arrow',
        size: 0.5
    }
});

var url = Ogma.utils.pixelRatio() === 2 ? // retina displays
    'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}@2x.png' :
    'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png';

// Initialize ogma with the data
var ogma = new Ogma({
    container: 'graph-container',
    settings: {
        geo: {
            tileUrlTemplate: url, // indicates from which server the
tiles must be retrieved
            sizeZoomReferential: 5, // Paris will be displayed with a
radius of 8 pixels on the screen if the geographical zoom is 5
            attribution: '<div class="attribution">Map data © <a target=
"_blank" href="http://osm.org/copyright">OpenStreetMap contributors</a></div>'
        },
        texts: {
            nodeFontSize: 20,
            edgeFontSize: 15,
            nodeBackgroundColor: '#fff',
        }
    },
    graph: {
        nodes: nodes,
        edges: edges
    }
});

ogma.geo.enable();

var pathNodes = ogma.pathfinding.dijkstra(startNode, endNode);
if (pathNodes) {
    var ids = pathNodes.map(function (node) {
        return node.id
    });

    // Color the path
    for (var i = 0; i < pathNodes.length; i++) {
        pathNodes[i].color = '#86315b';
        pathNodes[i].size = 2;

        ogma.topology.getAdjacentEdges(pathNodes[i]).forEach(function
(edge) {
            if (ids.indexOf(edge.source) !== -1 && ids.indexOf(edge.
target) !== -1 && ids.indexOf(edge.source) < ids.indexOf(edge.target)) {
                edge.color = '#86315b';
                edge.size = 0.4
            }
        });
    }
}

document.getElementById('n').textContent = 'nodes: ' + ogma.graph.nodes.length + ' ;
edges: ' + ogma.graph.edges.length;
}

```

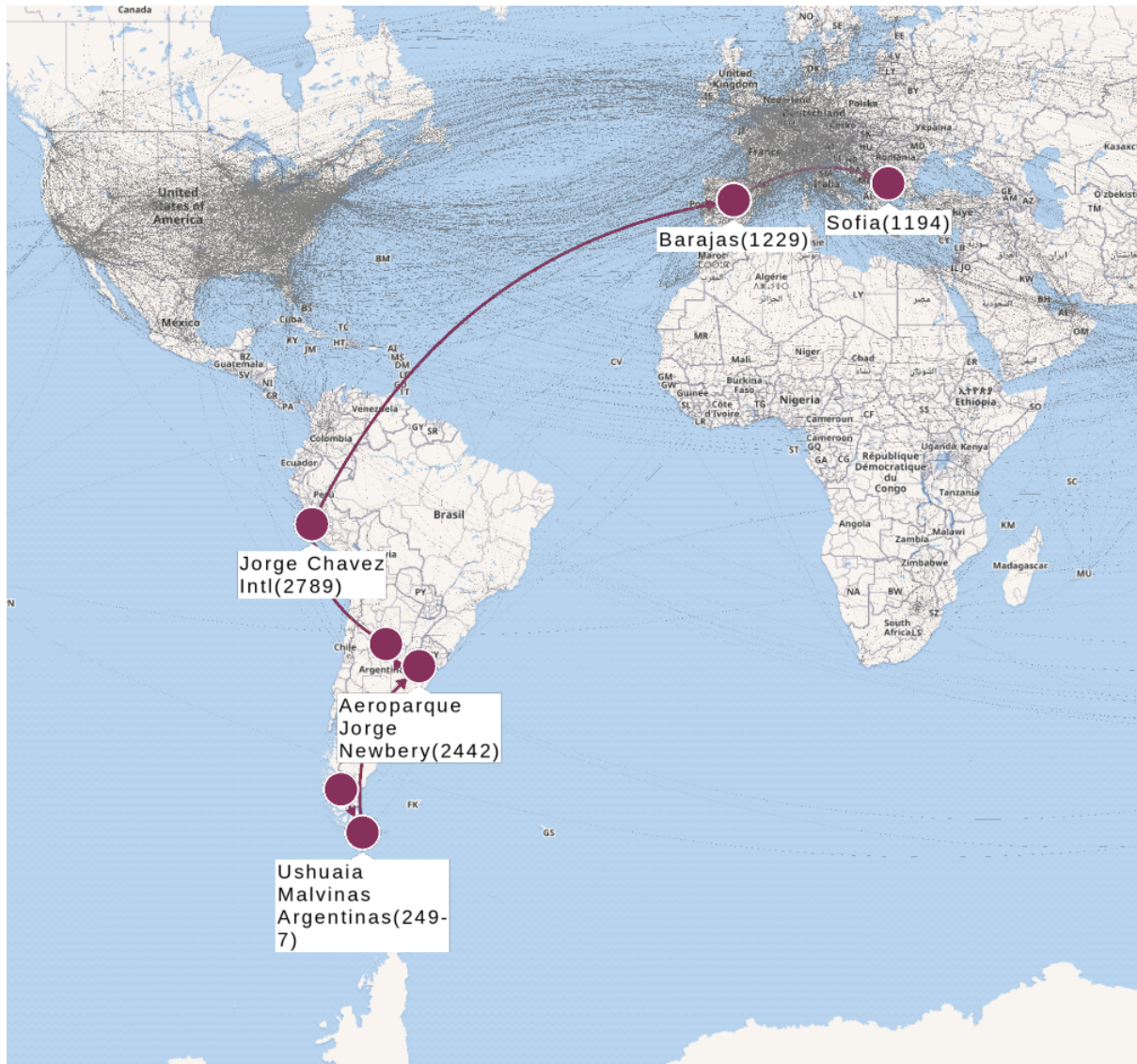
```

}}

</script>
</body>
</html>

```

Which produces the following graph:



7.8.4 Common function to visualize GraphDB Data

The commons.js file used by all demos

```

var stringToColour = function(str) {
  var hash = 0;
  for (var i = 0; i < str.length; i++) {
    hash = str.charCodeAt(i) + ((hash << 5) - hash);
  }
  var colour = '#';
  for (var i = 0; i < 3; i++) {
    var value = (hash >> (i * 8)) & 0xFF;
    colour += ('00' + value.toString(16)).substr(-2);
  }
}

```

```

    return colour;
}

var getLocalName = function(str) {
    return str.substr(Math.max(str.lastIndexOf('/'), str.lastIndexOf('#')) + 1);
}

var getPrefix = function(str) {
    return str.substr(0, Math.max(str.lastIndexOf('/'), str.lastIndexOf('#')));
}

var convertData = function(data) {
    var mapped = _.map(data, function(value, subject) {
        return _.map(value, function(value1, predicate) {
            return _.map(value1, function(object) {
                return [
                    subject,
                    predicate,
                    object.value
                ]
            })
        })
    });

    // Convert graph json to array of triples
    var triples = _.reduce(mapped, function(memo, el) {
        return memo.concat(el)
    }, []);
    triples = _.reduce(triples, function(memo, el) {
        return memo.concat(el)
    }, []);
    return triples;
}

// The RDFRank, nodes size is calculated according to RDFRank
var rankPredicate = "http://www.ontotext.com/owlim/RDFRank#hasRDFRank";

// Get type for a node to color nodes of the same type with the same color
var typePredicate = "http://www.openrdf.org/schema/sesame#directType";

// The location of a graphdb repo endpoint
var graphDBRepoLocation = 'http://factforge.net/repositories/ff-news';

```

Learn more about [linkurious](#) and [ogma](#).

7.9 Create custom graph view over your RDF data

What's in this document?

- *How it works?*
- *World airport, airline and route data*
 - *Data model*
 - *Configured queries*
- *Springer Nature SciGraph*

RDF is the most popular format for exchanging semantic data. Unlike, the logical database models the ontologies are optimised to represent the knowledge in a particular business domain correctly. This means that often their structure is verbose, includes abstract entities to express OWL axioms, contains implicit statements and complex N-ary relationship with provenance information. Graph View is a user interface optimised for mapping knowledge base models to simpler edge and vertex models configured by a list of SPARQL queries.

7.9.1 How it works?

In GraphDB 8.3 release, 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 colours. 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 ?image binding is found in the results the value is used as an image source.

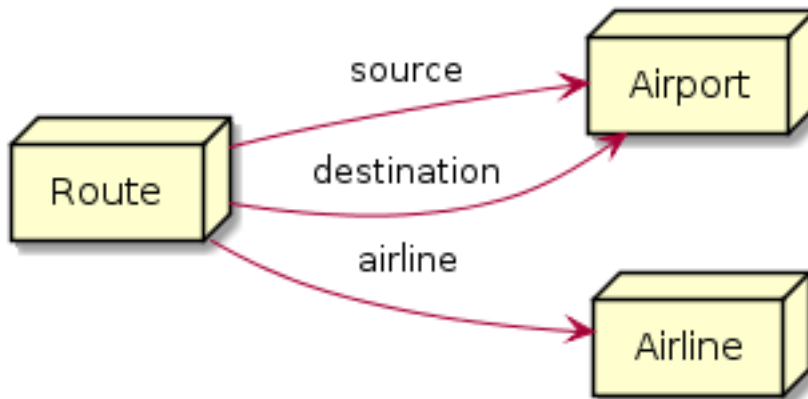
By using these four queries, the user may override the default configuration and adapt the knowledge base visualisation to display:

- Integrate custom ontology schema and what's the preferred label;
- Hide provenance or another meta-data 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.

7.9.2 World airport, airline and route data

[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 added manually. Significant revisions and additions 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 integrated with the rest of linked open data cloud.

7.9.2.1 Data model

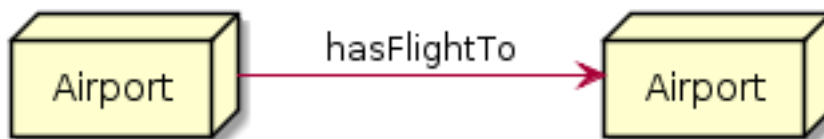


All OpenFlight CSV files are converted by using OntoRefine. Import the `airports.ttl` dataset which contains the data in RDF.

7.9.2.2 Configured queries

Find how airports are connected with flights

Lets find how are the airports connected by skipping the route relation and model a new relation `hasFlightTo`.



In SPARQL this can be done with the following query:

```

PREFIX onto: <http://www.ontotext.com/>
construct {
  ?source onto:hasFlightTo ?destination .
} where {
  ?flight <http://openflights.org/resource/route/sourceId> ?source .
  ?flight <http://openflights.org/resource/route/destinationId> ?destination
}
  
```

Using the new Visual button in the SPARQL editor we can now see the results of this query as a visual graph. But what if we want to save it and also expand more airports? 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:

```

PREFIX onto: <http://www.ontotext.com/>
construct {
  ?node onto:hasFlightTo ?destination .
} where {
  ?flight <http://openflights.org/resource/route/sourceId> ?node .
  ?flight <http://openflights.org/resource/route/destinationId> ?destination .
} limit 100
  
```

You may want to select each time a different airport to start from by making the starting point a search box.

Find which airlines fly to airports

The power of the visual graph is that on top of the same data we can create multiple Graph views. Let's create a new one using the following query.

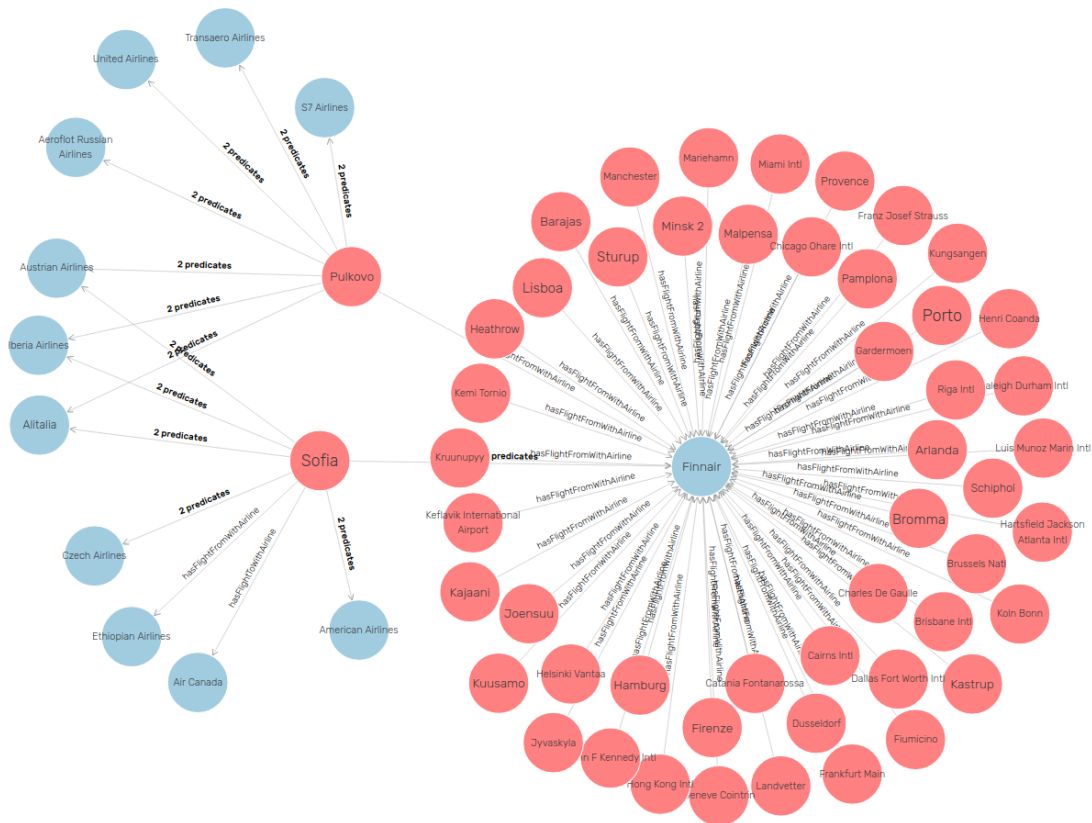
```
PREFIX onto: <http://www.ontotext.com/>
construct {
  ?airport onto:hasFlightFromWithAirline ?airline .
} where {
  ?route <http://openflights.org/resource/route/sourceId> ?airport .
  ?route <http://openflights.org/resource/route/airlineId> ?airline .
} limit 100
```

And let's create a visual graph with the following expand query:

```
# Note that ?node is the node you clicked and must be used in the query
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
PREFIX onto: <http://www.ontotext.com/>

CONSTRUCT {
  # The triples that will be added to the visual graph when you expand airports
  ?node onto:hasFlightFromWithAirline ?airline1 .
  ?node onto:hasFlightToWithAirline ?airline2 .

  # The triples to be added when you expand airlines
  ?airport1 onto:hasFlightFromWithAirline ?node .
  ?airport2 onto:hasFlightToWithAirline ?node .
} WHERE {
  {
    # Incoming flights for airport
    ?route <http://openflights.org/resource/route/sourceId> ?node .
    ?route <http://openflights.org/resource/route/airlineId> ?airline1 .
  } UNION {
    # Outgoing flights for airport
    ?route <http://openflights.org/resource/route/destinationId> ?node .
    ?route <http://openflights.org/resource/route/airlineId> ?airline2 .
  } UNION
  {
    # Incoming flights for airline
    ?route <http://openflights.org/resource/route/sourceId> ?airport1 .
    ?route <http://openflights.org/resource/route/airlineId> ?node .
  } UNION {
    # Outgoing flights for airline
    ?route <http://openflights.org/resource/route/destinationId> ?airport2 .
    ?route <http://openflights.org/resource/route/airlineId> ?node .
  }
}
```

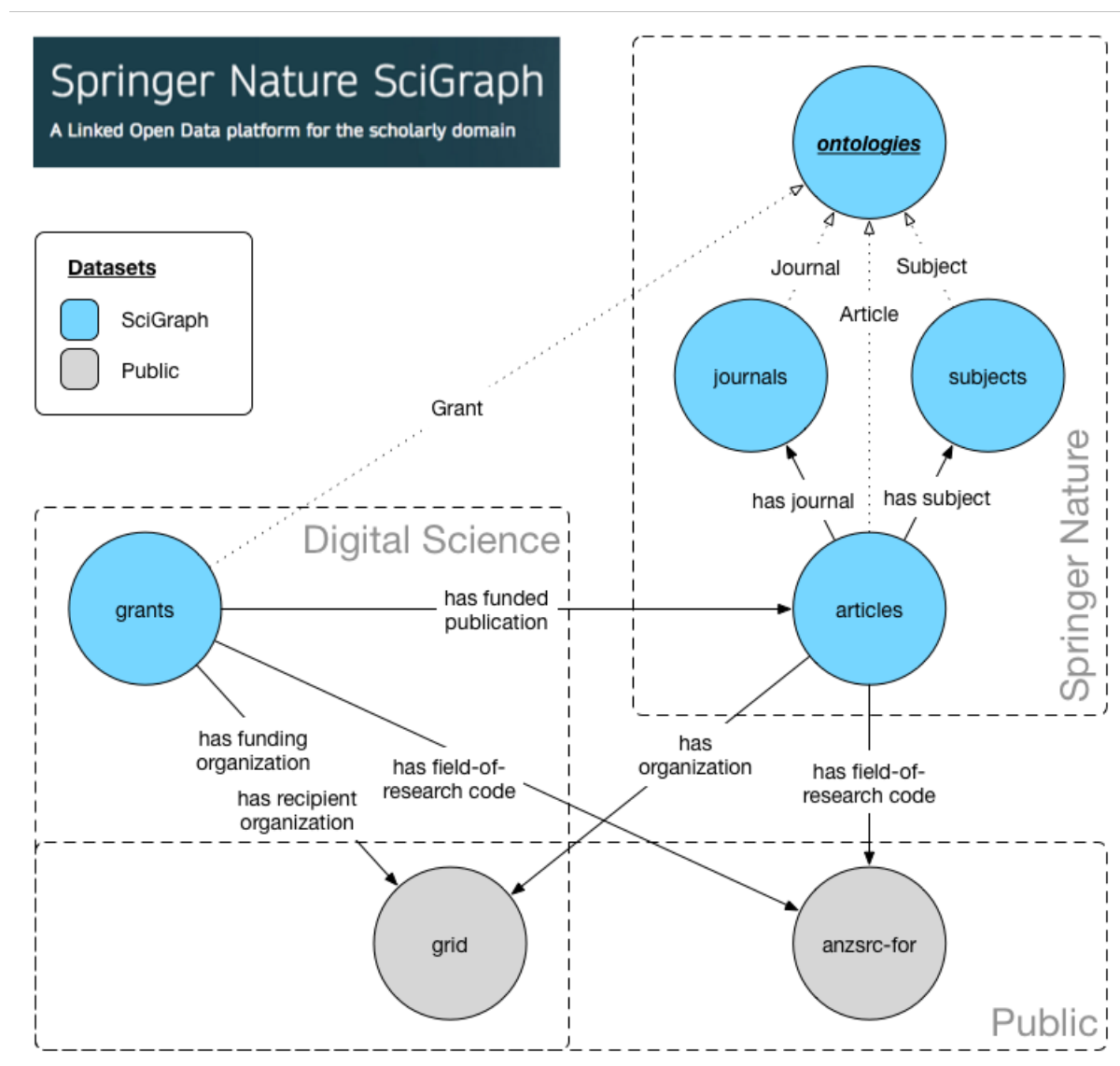



7.9.3 Springer Nature SciGraph

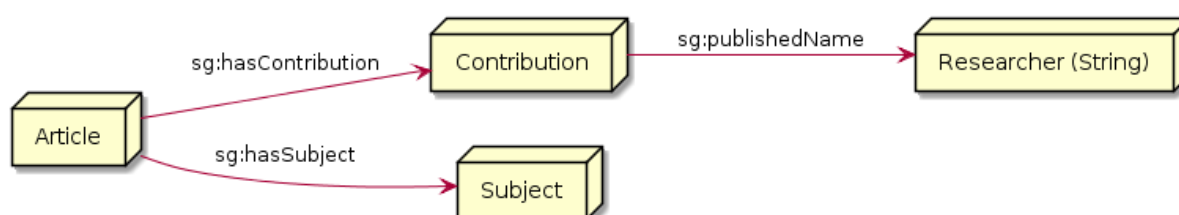
SciGraph is a linked Open Data platform for the scholarly domain. The dataset aggregates data sources from Springer Nature and key partners from the scholarly domain. It collates information from across the research landscape, for example, funders, research projects, conferences, affiliations and publications.

7.9.3.1 Data model

This is the full data model...



but let's say we are only interested in articles, contributions and subjects.



From this we can say that a researcher contributes to a subject and create a virtual URI for the researcher since it is a Literal.



Find researchers that contribute to the same subjects

We don't have a URI for a researcher. How can we search for researchers?



Navigate to *Setup* -> *Autocomplete* and add `sg:publishedName` predicate and as a result we will find contributions by given names in the search box.

Now let's create the graph config. We need to configure expansion for contribution since this is our starting point, for subjects and for researchers.

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX onto: <http://www.ontotext.com/>
PREFIX : <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 .
    }
}
```

Okay. But 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 to fetch some researchers and subjects.

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX onto: <http://www.ontotext.com/>
PREFIX : <http://www.grid.ac/ontology/>
construct {
    ?researcherNameUri2 onto:isContributorFor ?node .
}
where {
```

```
{
  BIND( IRI(CONCAT("http://www.ontotext.com/", REPLACE(STR(?researcherName), " ", "_"))) as ?
  ↳researcherNameUri2)
  ?contribution a sg:Contribution .
  ?contribution sg:publishedName ?researcherName .
  ?article sg:hasContribution ?contribution .
  ?article sg:hasSubject ?node .
}
} limit 100
```

But the nodes in our graph are all the same since they don't have RDF types. Now let's configure how the types of the nodes are obtained.

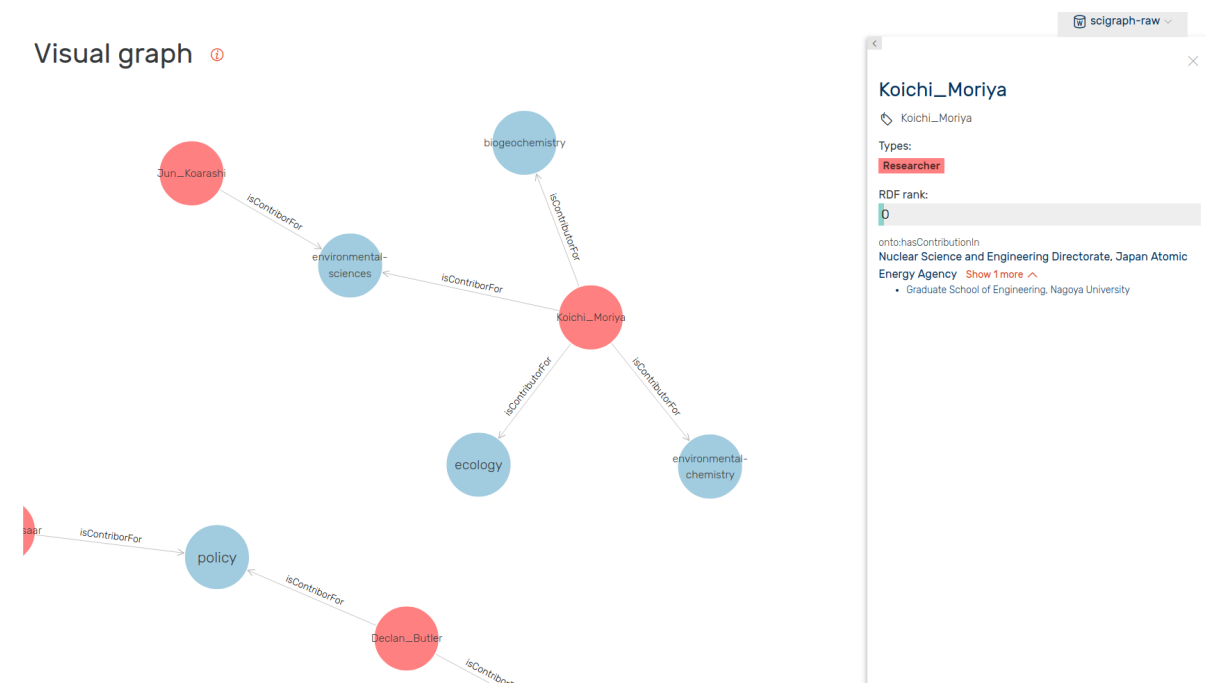
```
PREFIX sesame: <http://www.openrdf.org/schema/sesame#>

SELECT distinct ?type {
  # BIND (<http://www.ontotext.com/S._R._Arnold> as ?node)
  # # Get node type
  OPTIONAL {?node ?p ?o}
  BIND(IF (strStarts(STR(?node), "http://www.ontotext.com/"), "Researcher", "Subject") as ?type)
} ORDER BY ?type
```

Cool. But what if I want to see additional data for each node, i.e which university has a researcher contribution for:

```
PREFIX sg: <http://www.springernature.com/scigraph/ontologies/core/>
SELECT distinct ?property ?value where {
  # BIND (<http://www.ontotext.com/Kevin_J._Gaston> as ?node)
  BIND (<http://www.ontotext.com/hasContributionIn> as ?property)
  BIND( REPLACE(REPLACE(STR(?node), "_", " "), "http://www.ontotext.com/", "") as ?researcherName)
  optional {?node ?p ?o}
  ?contribution sg:publishedName ?researcherName .
  ?contribution sg:hasAffiliation ?affiliation .
  ?affiliation sg:publishedName ?value .
} limit 100
```

Visual graph



7.10 GraphDB System Statements

System statements are used as SPARQL pragmas specific to GraphDB. They are ways to alter the behaviour 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 `<http://www.ontotext.com>` and `<http://www.ontotext.com/owlim/system#>`, respectively.

7.10.1 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
}
```

- `onto:implicit` - the graph contains statements inferred via the repository's ruleset, located in the default graph.
- `onto:explicit` - the graph contains statements inserted in the database by the user, located in the default graph.
- `onto:readonly` - the graph contains schematic statements - i.e., the statements which define the repository's ruleset.
- `onto:readwrite` - the graph contains non-schematic statements - i.e., all statements beside the ones in the ruleset.
- `onto:count` - a pseudo graph that forces a count of the results of the query and returns it as the result.
- `onto:disable-sameAs` - a pseudo graph that disables the default behaviour of expanding the `sameAs` nodes of the query result.
- `onto:distinct` - a pseudo graph that makes the query behave as if the `DISTINCT` keyword was used.
- `onto:skip-redundant-implicit` - a pseudo graph that forces a check if a statement is already explicitly present in the result set and does not return implicit (inferred) versions of the same triple.
- `onto:merge` - specifies that for the given triple pattern merge join should be used. Should be used with a `GRAPH` clause, rather than `FROM`. Merge join is the process of intersecting the current triple pattern with the partial result set which has been already accumulated. They are suitable when there are different collections sharing one or more variables with collection sizes that are relatively equivalent.
- `onto:hash` - specifies that for the given triple pattern hash join should be used. Should be used with a `GRAPH` clause, rather than `FROM`. It performs an intersection between two sets of results by utilizing a hash table.
- `onto:explain` - a pseudo graph that returns the query optimization plan.
- `onto:commitStatistics` - if enabled, logs commit statistics every 30 seconds.
- `sys:statistics` - a pseudo graph that forces the usage of the repository statistics in `COUNT` queries, instead of properly counting the results when the `WHERE` clause consists of only one statement pattern. This speeds up the counting operation for simple queries, but it can produce wrong counts where results are manipulated, for example, by `owl:sameAs` expansion.
- `onto:retain-bind-position` - Doesn't allow `BIND` to move freely. Its position is preserved as the original one, relative to the vars which were before it and the vars which were after it.

7.10.2 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 {
  _:b sys:addRuleset "owl-horst-optimized" .
  _:b sys:defaultRuleset "owl-horst-optimized" .
  _:b sys:reinfer _:b .
}
```

- `sys:schemaTransaction` - allows for axiom insertion and removal, changing the ruleset.
- `sys:reinfer` - forces full inference recomputation.
- `sys:turnInferenceOn` - switches inferences on.
- `sys:turnInferenceOff` - disables inference. This will not remove previously inferred statements.
- `sys:addRuleset` - adds a ruleset.
- `sys:removeRuleset` - removes a ruleset.
- `sys:defaultRuleset` - refers to the default ruleset. Can be used to fetch it or change it.
- `sys:currentRuleset` - refers to the current ruleset. Can be used to fetch it or change it.
- `sys:listRulesets` - lists the currently installed rulesets.
- `sys:renameRuleset` - renames a ruleset.
- `sys:exploreRuleset` - retrieves a ruleset's text, if any.
- `sys:consistencyCheckAgainstRuleset` - checks data consistency against a given ruleset.
- `onto:replaceGraph` - Sets a replacement graph. The content of that graph will be replaced with the incoming update. Multiple graphs may be provided by multiple calls with this predicate.
- `onto:replaceGraphPrefix` - Sets a prefix for replacement graphs. All graphs whose IRIs start with the prefix will be replaced with the incoming update. Multiple prefixes may be provided by multiple calls with this predicate.

7.11 Time Functions Extensions

Beside the standard SPARQL functions related to time, GraphDB offers several additional functions, allowing users to do more with their time data. Those are implemented within the same namespace as standard math functions, `<http://www.ontotext.com/sparql/functions/>`. The default prefix for the functions is `ofn`.

7.11.1 Period extraction functions

The first group of functions is related to accessing particular parts of standard duration literals. For example, the expression `2019-03-24T22:12:29.183+02:00` - `"2019-04-19T02:42:28.182+02:00"` will produce the following duration literal: `-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	Description	Expected return value
<code>ofn:years-from-duration</code>	Return the “years” part of the duration literal.	0
<code>ofn:months-from-duration</code>	Returns the “months” part of the duration literal.	0
<code>ofn:days-from-duration</code>	Returns the “days” part of the duration literal.	25
<code>ofn:hours-from-duration</code>	Returns the “hours” part of the duration literal.	4
<code>ofn:minutes-from-duration</code>	Returns the “minutes” part of the duration literal.	29
<code>ofn:seconds-from-duration</code>	Returns the “seconds” part of the duration literal.	58
<code>ofn:millis-from-duration</code>	Returns the “milliseconds” part of the duration literal.	999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
  bind (ofn:millis-from-duration("-P0Y0M25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```

7.11.2 Period transformation functions

The second group of functions is related to transforming a standard duration literal. This reduces the need for performing mathematical transformations on the input date. The functions take a period as input and output `xsd:long`.

Note: The transformation is performed with no fractional components. For example, if transformed, the duration literal we’ve 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:

Function	Description	Expected return value
<code>ofn:asWeeks</code>	Returns the duration of the period as weeks.	-3
<code>ofn:asDays</code>	Returns the duration of the period as days.	-25
<code>ofn:asHours</code>	Returns the duration of the period as hours.	-604
<code>ofn:asMinutes</code>	Returns the duration of the period as minutes.	-36269
<code>ofn:asSeconds</code>	Returns the duration of the period as seconds.	-2176198
<code>ofn:asMillis</code>	Returns the duration of the period as milliseconds.	-2176198999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
  bind (ofn:asMillis("-P0Y0M25DT4H29M58.999S"^^xsd:dayTimeDuration) as ?result)
}
```

7.11.3 Durations Expressed in Certain Units

The third group of functions eliminates the need for computing a difference between two dates when a transformation will be necessary, essentially combining the mathematical operation of subtracting two dates with a transformation. It is more efficient than performing an explicit mathematical operation between two date literals, for example: "2019-03-24T22:12:29.183+02:00" - "2019-04-19T02:42:28.182+02:00" and then using a transformation function. The functions take two dates as input and output integer literals.

Note: Regular SPARQL subtraction can return negative values, as evidenced by the negative duration literal used in the example. However, comparisons are only positive. So, comparison isn't an exact match for a subtraction followed by transformation. If one of the timestamps has timezone but the other doesn't, the result is ill-defined.

The following table describes the functions that are implemented and gives example results, assuming the date literals "2019-03-24T22:12:29.183+02:00" and "2019-04-19T02:42:28.182+02:00" are passed to them. Note that the return values are positive:

Function	Description	Expected return value
ofn:weeksBetween	Returns the duration between the two dates as weeks.	3
ofn:daysBetween	Returns the duration between the two dates as days.	25
ofn:hoursBetween	Returns the duration between the two dates as hours.	604
ofn:minutesBetween	Returns the duration between the two dates as minutes.	36269
ofn:secondsBetween	Returns the duration between the two dates as seconds.	2176198
ofn:millisBetween	Returns the duration between the two dates as milliseconds.	2176198999

An example query using a function from this group would be:

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX ofn:<http://www.ontotext.com/sparql/functions/>
SELECT ?result {
  bind (ofn:millisBetween("2019-03-24T22:12:29.183+02:00"^^xsd:dateTime, "2019-04-19T02:42:28.182+02:00"^^xsd:dateTime) as ?result)
}
```

7.11.4 Arithmetic operations

The fourth group of functions includes operations such as: adding duration to a date; adding dayTimeDuration to a dateTime; adding time duration to a time; comparing durations. This is a SPARQL extension, more details about which are available at <https://www.w3.org/TR/sparql11-query/#operatorExtensibility>.

7.12 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.

7.12.1 Module 1: RDF & RDFS

RDF is a standardised 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.

7.12.2 Module 2: SPARQL

SPARQL is a SQL-like query language for RDF data. It is recognised 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.

7.12.3 Module 3: Ontology

This module looks at Ontologies: what is 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.

7.12.4 Module 4: GraphDB Installation

This video guides you through five steps in setting up your GraphDB: from downloading and deploying war files to your Tomcat Application Server, through launching Workbench, to final creation of a database and inserting and selecting data in it. Our favourite example from The Flintstones is available here as data for you to start with.

7.12.5 Module 5: Performance Tuning & Scalability

This module provides information on how to configure GraphDB for optimal performance and scalability. The size of datasets and the specific use cases benefit from different GraphDB memory configurations.

Watch this video to learn more about the four elements you can control as well as how to use GraphDB configuration tool. Tips about memory dedication during loading time and normal operation are provided as well.

7.12.6 Module 6: GraphDB Workbench & RDF4J

GraphDB Workbench is a web-based administration tool that allows you to manage GraphDB repositories, load and export data, monitor query execution, developing and executing queries, managing connectors and users. In this video we provide brief overview of the main functionality that you'll be using most of the time.

7.12.7 Module 7: Loading Data

Data is the most valuable asset and GraphDB is designed to store and enhance it. This module shows you how to use GraphDB Workbench to load individual files and bulk data from directories. For huge datasets we recommend speeding up the process by using Parallel bulk loader.

7.12.8 Module 8: Rule Set & Reasoning Strategies

This module outlines the reasoning strategies (how to get new information from your data) as well as the rule set 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 optimisation e.g. using `owl:sameAs`.

7.12.9 Module 9: Extensions

This module presents three extensions that empower GraphDB queries

RDFRank calculates connectives of nodes – similar to well known PageRank algorithm.

Geo-spatial queries extracts data placed in rectangles, polygons and circles.

Full text search provides faster access to textual data based on Apache Lucene, Solr and ElasticSearch.

7.12.10 Module 10: Troubleshooting

This module covers troubleshooting some common issues. These issues include both installation and operational issues. Installation issues covered include: Workbench, Lucene, Informatiq and custom rule files. Operational issues covered include: statement counts, deleting statements and socket timeouts.

7.13 FAQ

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.

How do I attach a location? Go to Admin/Locations and Repositories. Click the Attach location button. For a local location you need to provide the absolute pathname to a directory and for a remote location you need to provide a URL through which the server running the Workbench can see the remote GraphDB instance.

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.

How do I create a repository? Go to Admin/Locations and Repositories. Activate an existing location or add a new one, then click the Create repository button.

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 Admin/Cluster management, where you'll see a visual representation of repositories and each cluster. Drag and drop workers onto masters to connect them.

The GraphDB Developer Hub is meant as the central point for the GraphDB Developer Community. It is meant 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 modelling with RDF\(S\)*](#)
- [*SPARQL*](#)
- [*Ontologies*](#)
- [*Inference*](#)

If you are already familiar with RDF or you are eager to start programming please refer to:

- [*Programming with GraphDB*](#)
- [*Extending GraphDB Workbench*](#)
- [*Workbench 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*](#)

REFERENCES

8.1 Introduction to the Semantic Web

What's in this document?

- *Resource Description Framework (RDF)*
 - *Uniform Resource Identifiers (URIs)*
 - *Statements: Subject-Predicate-Object Triples*
 - *Properties*
 - *Named graphs*
- *RDF Schema (RDFS)*
 - *Describing classes*
 - *Describing properties*
 - *Sharing vocabularies*
 - *Dublin Core Metadata Initiative*
- *Ontologies and knowledge bases*
 - *Classification of ontologies*
 - *Knowledge bases*
- *Logic and inference*
 - *Logic programming*
 - *Predicate logic*
 - *Description logic*
- *The Web Ontology Language (OWL) and its dialects*
 - *OWL DLP*
 - *OWL Horst*
 - *OWL2 RL*
 - *OWL Lite*
 - *OWL DL*
- *Query languages*
 - *RQL, RDQL*
 - *SPARQL*

- *SeRQL*
- *Reasoning strategies*
 - *Total materialization*
- *Semantic repositories*

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 generalisation 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. 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`, if used as a URL (Web link) is expected to take us to a Web page of the site providing information about the site owner, the person John Doe. The same string can however be used simply to identify that person on the Web (URI) irrespective of whether such a page exists or not.

Thus URI schemes can be used not only for Web locations, but also for such diverse objects as telephone numbers, ISBN numbers, and geographic locations. In general, we assume that a URI is the identifier of a resource and can be used as either the subject or the object of a statement. Once the subject is assigned a URI, it can be treated as a resource and further statements can be made about it.

This idea of using URIs to identify ‘things’ and the relations between them is important. This approach goes some way towards a global, unique naming scheme. The use of such a scheme greatly reduces the homonym problem that has plagued distributed data representation in the past.

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:

“[subject] the web site `www.johndoesite.com` [predicate] has a creator [object] called John Doe.”

This statement emphasises the fact that in order to describe something, there has to be a way to name or identify a number of things:

- the thing the statement describes (Web site “`www.johndoesite.com`”);
- a specific property (“creator”) of the thing the statement describes;
- the thing the statement says is the value of this property (who the owner is).

The respective RDF terms for the various parts of the statement are:

- the subject is the URL “`www.johndoesite.com`”;
- the predicate is the expression “has creator”;
- the object is the name of the creator, which has the value “John Doe”.

Next, each member of the subject-predicate-object triple should be identified using its URI, e.g.:

- the subject is `http://www.johndoesite.com`;
- the predicate is `http://purl.org/dc/elements/1.1/creator` (this is according to a particular RDF Schema, namely, the Dublin Core Metadata Element Set);

- the object is `http://www.johndoesite.com/aboutme` (which may not be an actual web page).

Note that in this version of the statement, instead of identifying the creator of the web site by the character string “John Doe”, we used a URI, namely `http://www.johndoesite.com/aboutme`. An advantage of using a URI is that the identification of the statement’s subject can be more precise, i.e., the creator of the page is neither the character string “John Doe”, nor any of the thousands of other people with that name, but the particular John Doe associated with this URI (whoever created the URI defines the association). Moreover, since there is a URI to refer to John Doe, he is now a full-fledged resource and additional information can be recorded about him simply by adding additional RDF statements with John’s URI as the subject.

What we basically have now is the logical formula $P(x, y)$, where the binary predicate P relates the object x to the object y – we may also think of this formula as written in the form x, P, y . In fact, RDF offers only binary predicates (properties). If more complex relationships are to be defined, this is done through sets of multiple RDF triples. Therefore, we can describe the statement as:

```
<http://www.johndoesite.com> <http://purl.org/dc/elements/1.1/creator> <http://www.johndoesite.com/
↳aboutme>
```

There are several conventions for writing abbreviated RDF statements, as used in the RDF specifications themselves. This shorthand employs an XML *qualified name* (or *QName*) without angle brackets as an abbreviation for a full URI reference. A QName contains a prefix that has been assigned to a namespace URI, followed by a colon, and then a local name. The full URI reference is formed from the QName by appending the local name to the namespace URI assigned to the prefix. So, for example, if the QName prefix `foo` is assigned to the namespace URI `http://example.com/somewhere/`, then the QName “foo:bar” is a shorthand for the URI `http://example.com/somewhere/bar`.

In our example, we can define the namespace `jds` for `http://www.johndoesite.com` and use the Dublin Core Metadata namespace `dc` for `http://purl.org/dc/elements/1.1/`.

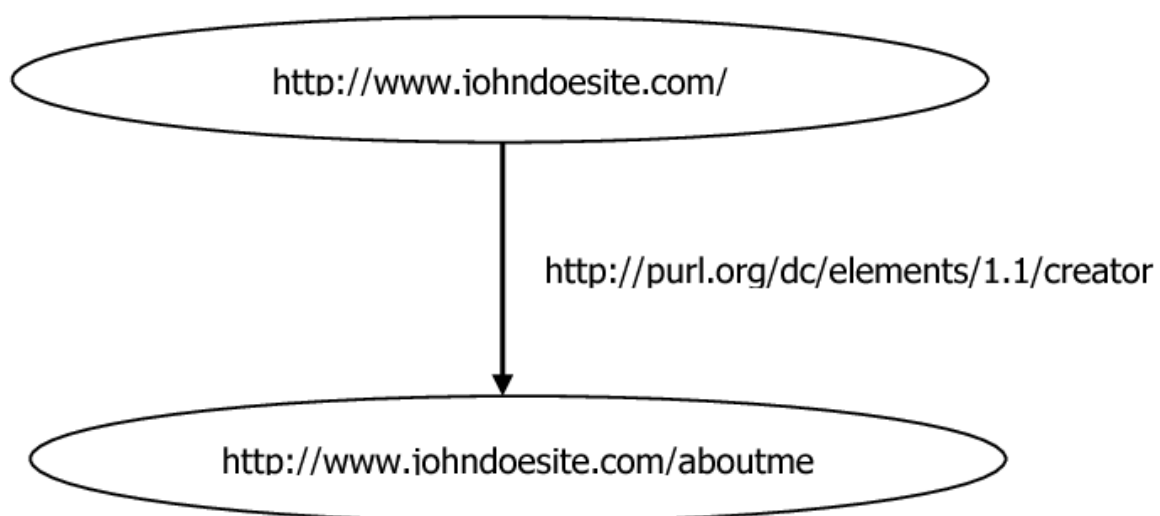
So, the shorthand form for the example statement is simply:

```
jds: dc:creator jds:aboutme
```

Objects of RDF statements can (and very often do) form the subjects of other statements leading to a graph-like representation of knowledge. Using this notation, a statement is represented by:

- a node for the subject;
- a node for the object;
- an arc for the predicate, directed from the subject node to the object node.

So the RDF statement above could be represented by the following graph:



This kind of graph is known in the artificial intelligence community as a ‘semantic net’.

In order to represent RDF statements in a machine-processable way, RDF uses mark-up languages, namely (and almost exclusively) the Extensible Mark-up Language (XML). Because an abstract data model needs a concrete syntax in order to be represented and transmitted, RDF has been given a syntax in XML. As a result, it inherits the benefits associated with XML. However, it is important to understand that other syntactic representations of RDF, not based on XML, are also possible. XML-based syntax is not a necessary component of the RDF model. XML was designed to allow anyone to design their own document format and then write a document in that format. RDF defines a specific XML mark-up language, referred to as RDF/XML, for use in representing RDF information and for exchanging it between machines. Written in RDF/XML, our example will look as follows:

```
<?xml version="1.0" encoding="UTF-16"?>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:jds="http://www.johndoesite.com/">

  <rdf:Description rdf:about="http://www.johndoesite.com/">
    <dc:creator rdf:resource="jds:aboutme">
  </rdf:Description>
</rdf:RDF>
```

Note: RDF/XML uses the namespace mechanism of XML, but in an expanded way. In XML, namespaces are only used for disambiguation purposes. In RDF/XML, external namespaces are expected to be RDF documents defining resources, which are then used in the importing RDF document. This mechanism allows the reuse of resources by other people who may decide to insert additional features into these resources. The result is the emergence of large, distributed collections of knowledge.

Also observe that the `rdf:about` attribute of the element `rdf:Description` is equivalent in meaning to that of an ID attribute, but it is often used to suggest that the object about which a statement is made has already been ‘defined’ elsewhere. Strictly speaking, a set of RDF statements together simply forms a large graph, relating things to other things through properties, and there is no such concept as ‘defining’ an object in one place and referring to it elsewhere. Nevertheless, in the serialised XML syntax, it is sometimes useful (if only for human readability) to suggest that one location in the XML serialisation is the ‘defining’ location, while other locations state ‘additional’ properties about an object that has been ‘defined’ elsewhere.

8.1.1.3 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.

8.1.1.4 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 generalisation 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 organised 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 specialised 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 characterise 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 `ex:weightInKg`, is defined with the following RDF statement:

```
ex: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:title>A Guide to Growing Roses</dc:title>
    <dc:description>Describes planting and nurturing rose bushes.
  </dc:description>
    <dc:date>2001-01-20</dc:date>
  </rdf:Description>
</rdf:RDF>
```

The same RDF statements in Notation-3:

```
@prefix dc: <[http://purl.org/dc/elements/1.1/]> .
@prefix rdf: <[http://www.w3.org/1999/02/22-rdf-syntax-ns#]> .

<http://media.example.com/audio/guide.ra> dc:creator "Mr. Dan D. Lion" ;
dc:title "A Guide to Growing Roses" ;
dc:description "Describes planting and nurturing rose bushes." ;
dc:date "2001-01-20" .
```

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 modelling 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 conceptualisation that they formalise: 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 modelled 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](#);

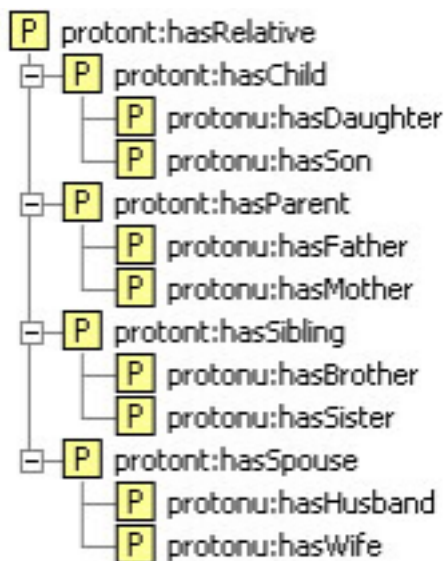
- Lexical ontologies: lexicons with formal semantics that define lexical concepts. We use ‘lexical concept’ here as some kind of a formal representation of the meaning of a word or a phrase. In Wordnet, for example, lexical concepts are modelled as synsets (synonym sets), while word-sense is the relation 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 conceptualisation. 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., modelling of concepts such as people, organisations, 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 specialisations of predicate logic. One reason for such specialised 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:

$A_1, \dots, A_n \rightarrow B$

where A_i and B are atomic formulas. In fact, there are two intuitive ways of reading such a rule:

- If $A_1, \dots, A_n$ are known to be true, then B is also true. Rules with this interpretation are referred to as ‘deductive rules’.
- If the conditions $A_1, \dots, A_n$ 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’.

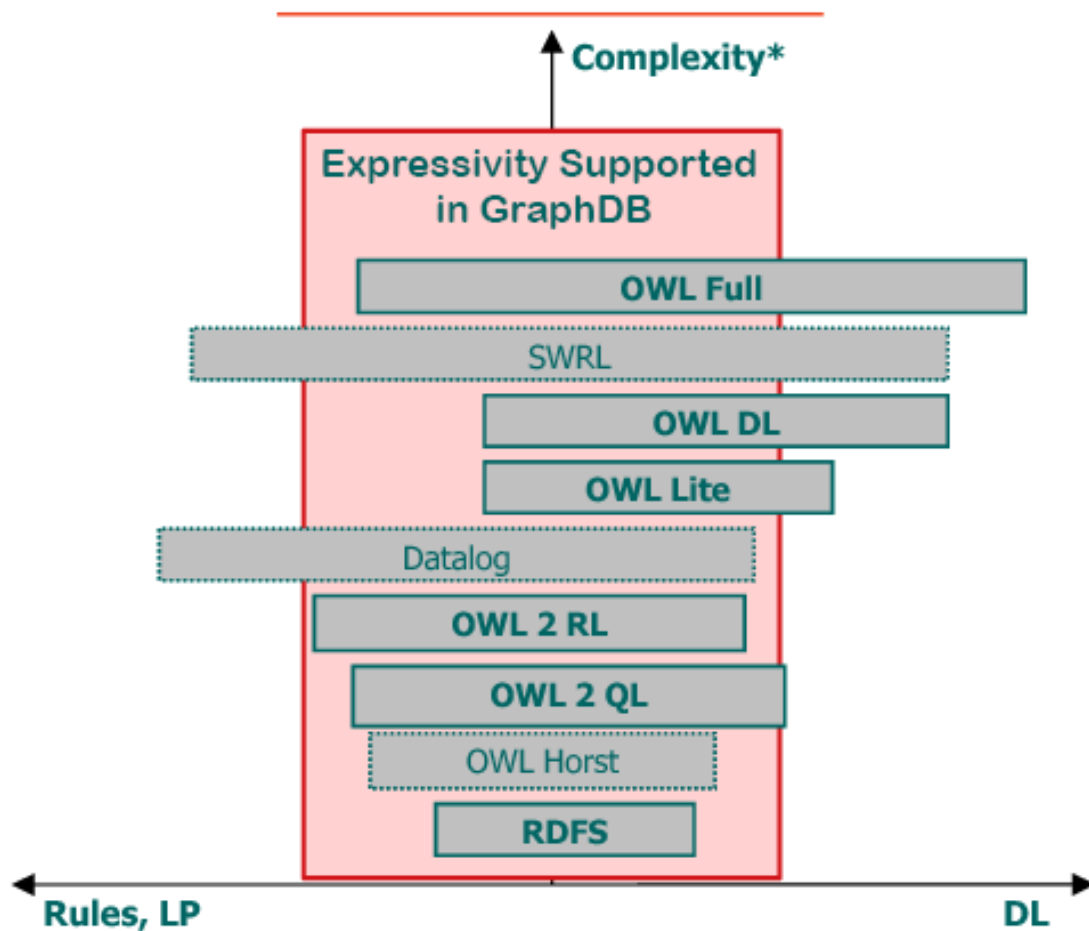
The family of DL system consists of many members that differ mainly with respect to the constructs they provide. Not all of the constructs can be found in a single DL system.

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

Naïve OWL Fragments 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 sublanguage 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 modelling 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 modeller’s perspective, there is freedom to use either OWL or rules (and associated tools and methodologies) for modelling purposes, depending on the modeller’s experience and preferences.
- 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 generalised 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, OWL 2 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 expressive power. It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2. This is achieved by defining a syntactic subset of OWL 2, which is amenable to implementation using rule-based technologies, and presenting a partial axiomatisation 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 sublanguage 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 sublanguage 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 an 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 standardised 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: Initialisation 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 materialisation. In order to avoid ambiguity with various partial materialisation 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 materialisation.

Advantages and disadvantages of the total materialisation:

- 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 materialisation, is that they can easily benefit from RDBMS-like query optimisation 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 optimisations are much more complex in the case of deductive query evaluation.

Total materialisation 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,

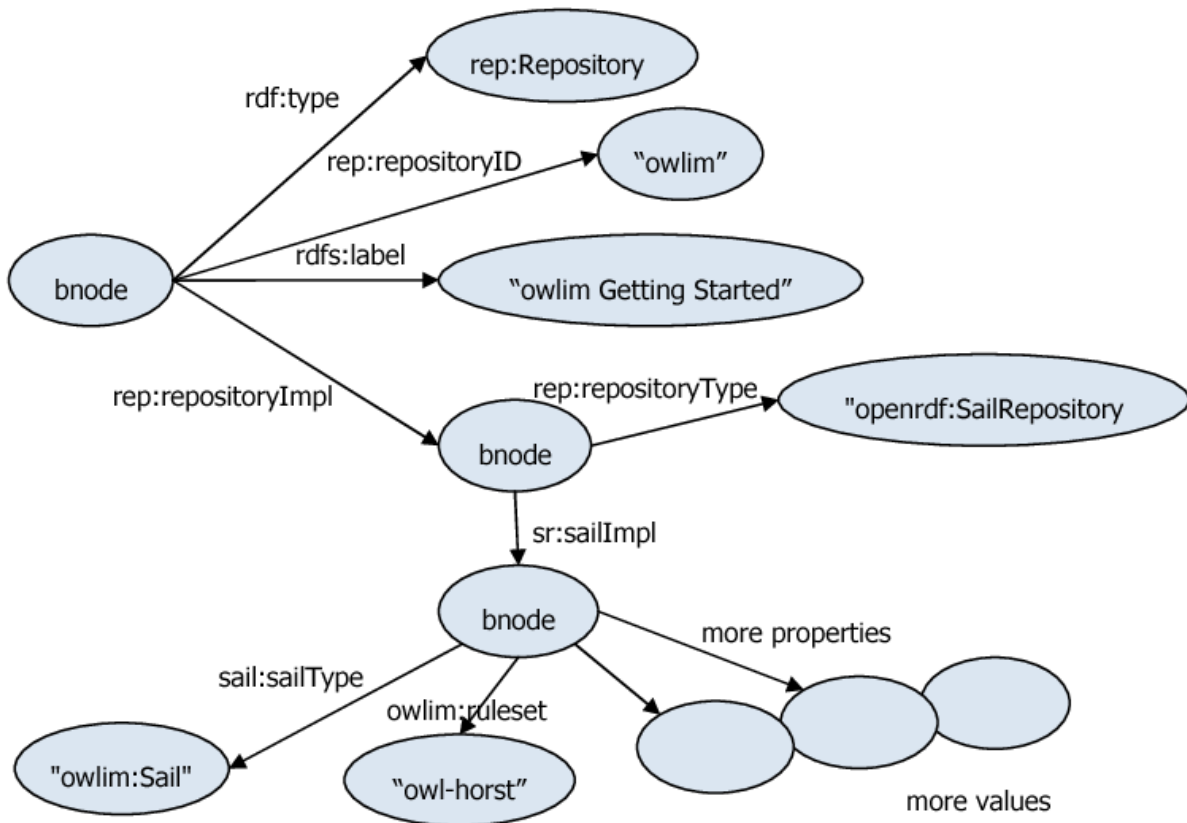
performance, intended application, etc. Introducing the term ‘semantic repository’ is an attempt to convey the core functionality offered by most of these tools. Semantic repositories can be used as a replacement for database management systems (DBMS), offering easier integration of diverse data and 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

Feature	GraphDB Free	GraphDB SE	GraphDB EE
Manage unlimited number of RDF statements	✓	✓	✓
Full <i>SPARQL 1.1 support</i>	✓	✓	✓
Deploy anywhere using Java	✓	✓	✓
100% compatible with RDF4J framework	✓	✓	✓
Ultra fast <i>forward-chaining reasoning</i>	✓	✓	✓
Efficient <i>retraction of inferred statements</i> upon update	✓	✓	✓
Full standard-compliant and optimised <i>rulesets</i> for RDFS, OWL 2 RL and QL	✓	✓	✓
Custom <i>reasoning</i> and <i>consistency checking rulesets</i>	✓	✓	✓
<i>Plugin API</i> for engine extension	✓	✓	✓
Support for <i>Geo-spatial indexing</i> & querying, plus <i>GeoSPARQL</i>	✓	✓	✓
<i>Query optimizer</i> allowing effective query execution	✓	✓	✓
<i>Workbench interface</i> to manage repositories, data, user accounts and access roles	✓	✓	✓
Lucene connector for full-text search	✓	✓	✓
Solr connector for full-text search	✗	✗	✓
Elasticsearch connector for full-text search	✗	✗	✓
High performance load, query and inference simultaneously	Limited to two concurrent queries	✓	✓
Automatic failover, synchronisation and load balancing to maximize cluster utilisation	✗	✗	✓
Scale out concurrent query processing allowing query throughput to scale proportionally to the number of cluster nodes	✗	✗	✓
Cluster elasticity remaining fully functional in the event of failing nodes	✗	✗	✓
Community support	✓	✓	✓
Commercial SLA	✗	✓	✓

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 `owlim: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.

The type of the repository is defined via the `rep:repositoryType` property and its value must be `owlim:MonitorRepository` to allow for custom Sail implementations (such as GraphDB) to be used in RDF4J 2.0. 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 `owlim: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:

```

rep:  <http://www.openrdf.org/config/repository#>
sr:   <http://www.openrdf.org/config/repository/sail#>
sail: <http://www.openrdf.org/config/sail#>
owlim: <http://www.ontotext.com/tree/owlim#>

```

All properties used to specify the GraphDB *configuration parameters* use the `owlim:` prefix and the local names match up with the *Configuration parameters*, e.g., the value of the `ruleset` parameter can be specified using the `http://www.ontotext.com/tree/owlim#ruleset` property.

8.4 Ontology mapping with `owl:sameAs` property

GraphDB `owl:sameAs` optimisation 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's 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 4 different datasets, you can use the following query on DBPedia to select everything about Sofia:

```
SELECT * {
  {
    <http://dbpedia.org/resource/Sofia> ?p ?o .
  }
  UNION
  {
    <http://data.nytimes.com/nytimes:N82091399958465550531> ?p ?o .
  }
  UNION
  {
    <http://sws.geonames.org/727011/> ?p ?o .
  }
  UNION
  {
    <http://rdf.freebase.com/ns/m/0ftjx> ?p ?o .
  }
}
```

Or you can even use a shorter one:

```
SELECT * {
  ?s ?p ?o
  FILTER (?s IN (
    <http://dbpedia.org/resource/Sofia>,
    <http://data.nytimes.com/nytimes:N82091399958465550531>,
    <http://sws.geonames.org/727011/>,
    <http://rdf.freebase.com/ns/m/0ftjx>))
}
```

As you can see, here Sofia appears with 4 different URIs, although they denote the same concept. Of course, this is a very simple query. Sofia has also relations to other entities in these datasets, such as Plovdiv, i.e., <[http://dbpedia.org/resource/Plovdiv]>, <[http://sws.geonames.org/653987/]>, <[http://rdf.freebase.com/ns/m/1aihge]>.

What's more, not only the different instances of one concept have multiple names but their properties also appear with many names. Some of them are specific for a given dataset (e.g., GeoNames has longitude and latitude, while DBPedia provides wikilinks) but there are class hierarchies, labels and other common properties used by most of the datasets.

This means that even for the simplest query you may have to write the following:

```
SELECT * {
  ?s ?p1 ?x .
  ?x ?p2 ?o .
  FILTER (?s IN (
    <http://dbpedia.org/resource/Sofia>,
    <http://data.nytimes.com/nytimes:N82091399958465550531>,
    <http://sws.geonames.org/727011/>,
    <http://rdf.freebase.com/ns/m/0ftjx>))
  FILTER (?p1 IN (
    <http://dbpedia.org/property/wikilink>,
    <http://sws.geonames.org/p/relatesTo>))
  FILTER (?p2 IN (
    <http://dbpedia.org/property/wikilink>,
    <http://sws.geonames.org/p/relatesTo>))
  FILTER (?o IN (<http://dbpedia.org/resource/Plovdiv>,
```

```
<http://sws.geonames.org/653987/>,
<http://rdf.freebase.com/ns/m/1aihge>))
}
```

But if you can say through rules and assertions that given URIs are the same, then you can simply write:

```
SELECT * {
  <http://dbpedia.org/resource/Sofia> <http://sws.geonames.org/p/relatesTo> ?x .
  ?x <http://sws.geonames.org/p/relatesTo> <http://dbpedia.org/resource/Plovdiv> .
}
```

If you link two nodes with `owl:sameAs`, the statements that appear with the first node's subject, predicate and object will be copied, replacing respectively the subject, predicate and the object that appear with the second node.

For example, given that `<[http://dbpedia.org/resource/Sofia]> owl:sameAs <[http://data.nytimes.com/N82091399958465550531]>` and also that:

```
<http://dbpedia.org/resource/Sofia> a <http://dbpedia.org/resource/Populated_place> .
<http://data.nytimes.com/N82091399958465550531> a <http://www.opengis.net/gml/_Feature> .
<http://dbpedia.org/resource/Plovdiv> <http://dbpedia.org/property/wikilink> <http://dbpedia.org/
↪resource/Sofia> .
```

then you can conclude with the given rules that:

```
<http://dbpedia.org/resource/Sofia> a <http://www.opengis.net/gml/_Feature> .
<http://data.nytimes.com/N82091399958465550531> a <http://dbpedia.org/resource/Populated_place> .
<http://dbpedia.org/resource/Plovdiv> <http://dbpedia.org/property/wikilink> <http://data.nytimes.
↪com/N82091399958465550531> .
```

The challenge with `owl:sameAs` is that when there are many 'mappings' of nodes between datasets, and especially when big chains of `owl:sameAs` appear, it becomes inefficient. `owl:sameAs` is defined as Symmetric and Transitive, so given that `A sameAs B sameAs C`, it also follows that `A sameAs A`, `A sameAs C`, `B sameAs A`, `B sameAs B`, `C sameAs A`, `C sameAs B`, `C sameAs C`. If you have such a chain with N nodes, then N^2 `owl:sameAs` statements will be produced (including the explicit $N-1$ `owl:sameAs` statements that produce the chain). Also, the `owl:sameAs` rules will copy the statements with these nodes N times, given that each statement contains only one node from the chain and the other nodes are not `sameAs` anything. But you can also have a statement `<S P O>` where `S sameAs Sx`, `P sameAs Py`, `O sameAs Oz`, where the `owl:sameAs` statements for `S` are K , for `P` are L and for `O` are M , yielding $K*L*M$ statement copies overall.

Therefore, instead of using these simple rules and axioms for `owl:sameAs` (actually 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 given rules are commented out in the PIE files and are left only as a reference.

8.5 Workbench User Interface

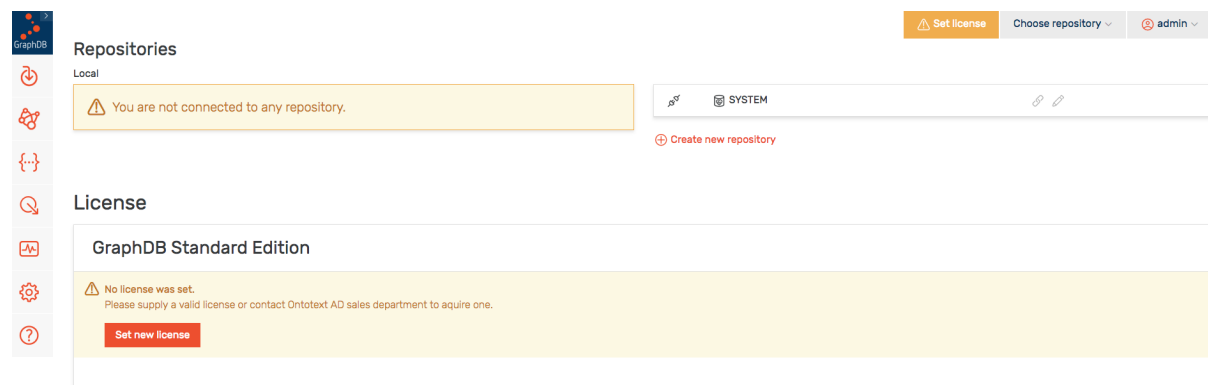
What's in this document?

- *Workbench Functionalities Descriptions*
- *Workbench configuration properties*

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

The Workbench layout consists of two main areas. The navigation area is on the left-hand side of the screen and it contains dropdown menus to all functionalities - Import, Explore, SPARQL, Export/Context, Monitor, Setup, and Help. The work area shows the tasks associated with the selected functionality. On the home page, it provides

easy access to some of the actions in the workbench such as set a license, attach location, create a repository, find a resource, query your data, etc. On the bottom of the page, you can see the license details, and in the footer - the versions of the various GraphDB components.



8.5.1 Workbench Functionalities Descriptions

Navigation Tab	Functionality Description
Import	• RDF => 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 <i>Text area</i> tab. Each import method supports different serialisation formats. • Tabular (OntoRefine) => Convert tabular data into RDF and import it into a GraphDB repository using simple SPARQL queries and a virtual endpoint. The supported formats are TSV, CSV, *SV, Excel (.xls and .xlsx), JSON, XML, RDF as XML, and Google sheet.
Explore	• 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 colour of the class with more incoming links. The links can be in both directions. • View resource => View and edit all RDF data related to a resource. • Visual graph => Explore your data graph in a visual way. Expand resources of interest to see their links.
SPARQL	• SPARQL => Query and update your data. Use any type of SPARQL query and click Run to execute it.
Export/Context	• Export => Export an entire repository or one or more graphs (contexts). Multiple formats are supported. • Context => Filter and manage the contexts (graphs) in a repository - inspect its triples, downloaded, or delete it.
Monitor	• Resources => Monitor the usage of various system resources, such as memory and CPU, for the currently active location. • Queries => Monitor all running queries in GraphDB. Any query can be killed by pressing the <i>Abort query</i> button.
426	Chapter 8. References
Setup	• Repositories => Manage repositories and connect to remote locations. A location represents

8.5.2 Workbench 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`):

Parameter	Description
<code>graphdb.workbench.cors.enable</code>	Enables cross-origin resource sharing. Default: <code>false</code>
<code>graphdb.workbench.cors.origin</code>	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. Default: <code>*</code>
<code>graphdb.workbench.maxConnections</code>	Sets the maximum number of concurrent connections to a remote GraphDB instance. Default: <code>200</code>
<code>graphdb.workbench.datadir</code>	Sets the directory where the workbench persistence data will be stored. Default: <code>\${user.home}/.graphdb-workbench/</code>
<code>graphdb.workbench.importDirectory</code>	Changes the location of the file import folder. Default: <code>\${user.home}/graphdb-import/</code>
<code>graphdb.workbench.maxUploadSize</code>	Sets the maximum upload size for importing local files. The value must be in bytes. Default: <code>200 MB</code>

8.6 SPARQL compliance

What’s in this document?

- *SPARQL 1.1 Protocol for RDF*
- *SPARQL 1.1 Query*
- *SPARQL 1.1 Update*
 - *Modification operations on the RDF triples:*
 - *Operations for managing graphs:*
- *SPARQL 1.1 Federation*
 - *Internal SPARQL federation*
- *SPARQL 1.1 Graph Store HTTP Protocol*
 - *URL patterns for this new functionality are provided at:*
 - *Methods supported by these resources and their effects:*
 - *Request headers:*
 - *Supported parameters for requests on indirectly referenced named graphs:*

GraphDB supports the following SPARQL specifications:

8.6.1 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.

8.6.2 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.

8.6.3 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` behaviour. For full details, see the [W3C SPARQL Update working group page](#).

8.6.3.1 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.

8.6.3.2 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.

8.6.4 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.

For example, to discover DBpedia resources about people who have the same names as those stored in a local repository, use the following query:

```

SELECT ?dbpedia_id
WHERE {
  ?person a foaf:Person ;
          foaf:name ?name .
  SERVICE <http://dbpedia.org/sparql> {
    ?dbpedia_id a dbpedia-owl:Person ;
                foaf:name ?name .
  }
}

```

It matches the first part against the local repository and for each person it finds, it checks the DBpedia SPARQL endpoint to see if a person with the same name exists 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, which have a similar `rdfs:label` to the `rdfs:label` of `<http://dbpedia.org/resource/Vaccination>` in the first SPARQL endpoint:

```

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

SELECT ?endpoint2_id {
  SERVICE <http://faraway_endpoint.org/sparql>
  {
    ?endpoint1_id rdfs:label ?l1 .
    FILTER( lang(?l1) = "en" )
  }
  SERVICE <http://remote_endpoint.com/sparql>
  {
    ?endpoint2_id rdfs:label ?l2 .
    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.

8.6.4.1 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 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 SPARQL federation* mechanism. Since this approach skips all HTTP communication overheads, it is more efficient.

For example, imagine that you have a GraphDB instance with two repositories - one called `my_concepts` with triples about concepts and another called `my_labels`, containing all label information.

To retrieve the corresponding label for each concept, you can execute the following query on the `my_concepts` repository:

```

SELECT ?id ?label
WHERE {
  ?id a ex:Concept .
  SERVICE <repository:my_labels> {
    ?id rdfs:label ?label.
  }
}

```

```
}  
}
```

The approach applied for DBpedia, SERVICE <http://localhost:7200/repositories/my_labels> is also valid, but is less efficient.

The *Internal SPARQL federation* page contains more details about it, including query parameters that can be used to control the behaviour of the federated query.

8.6.5 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.

8.6.5.1 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).

8.6.5.2 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.

8.6.5.3 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.

8.6.5.4 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.

8.7 Using math functions with SPARQL

GraphDB supports standard math functions to be used with SPARQL.

The following query summarizes all implemented math functions:

```
PREFIX ofn: <http://www.ontotext.com/sparql/functions/>

SELECT * {

  # acos
  # The arccosine function. The input is in the range[-1, +1]. The output is in the range [0, pi]
  ↪radians.
  BIND (ofn:acos(0.5) AS ?acos)

  # asin
  # The arcsine function. The input is in the range[-1, +1]. The output is in the range [-pi/2, pi/
  ↪2] radians.
  BIND (ofn:asin(0.5) AS ?asin)

  # atan
  # The arctangent function. The output is in the range (-pi/2, pi/2) radians.
  BIND (ofn:atan(1) AS ?atan)

  # atan2
  # The double-argument arctangent function (the angle component of the conversion from rectangular
  ↪coordinates
  # to polar coordinates), see Math.atan2().
  # The output is in the range [-pi/2, pi/2] radians.
  BIND (ofn:atan2(1, 0) AS ?atan2)

  # cbrt
  # The cubic root function.
  BIND (ofn:cbrt(2) AS ?cbrt)

  # copySign
  # Returns the first floating-point argument with the sign of the second floating-point argument,
  ↪see Math.copySign().
  BIND (ofn:copySign(2, -7.5) AS ?copySign)

  # cos
  # The cosine function. The argument is in radians.
  BIND (ofn:cos(1) AS ?cos)

  # cosh
  # The hyperbolic cosine function.
  BIND (ofn:cosh(1) AS ?cosh)

  # e
  # The E constant, the base of the natural logarithm.
  BIND (ofn:e() AS ?e)

  # exp
  # The exponent function, e^x.
  BIND (ofn:exp(2) AS ?exp)

  # expm1
  # The Math.expm1() function. Returns e^x - 1.
  BIND (ofn:expm1(3) AS ?expm1)

  # floorDiv
  # Returns the largest (closest to positive infinity) int value that is less than or equal to the
  ↪algebraic quotient.
```

```

# The arguments are implicitly cast to long.
BIND (ofn:floorDiv(5, 2) AS ?floorDiv)

# floorMod
# Returns the floor modulus of the int arguments. The arguments are implicitly cast to long.
BIND (ofn:floorMod(10, 3) AS ?floorMod)

# getExponent
# Returns the unbiased exponent used in the representation of a double.
# This means that we take N from the binary representation of X:  $X = 1 * 2^N + \{1|0\} * 2^{(N-1)} + \dots + \{1|0\} * 2^0$ ,
↪ + {1|0} * 2^0,
# i.e. the power of the highest non-zero bit of the binary form of X.
BIND (ofn:getExponent(10) AS ?getExponent)

# hypot
# Returns sqrt(x^2 + y^2) without intermediate overflow or underflow.
BIND (ofn:hypot(3, 4) AS ?hypot)

# IEEEremainder
# Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
BIND (ofn:IEEEremainder(3, 4) AS ?IEEEremainder)

# log
# The natural logarithm function.
BIND (ofn:log(4) AS ?log)

# log10
# The common (decimal) logarithm function.
BIND (ofn:log10(4) AS ?log10)

# log1p
# The Math.log1p() function.
# Returns the natural logarithm of the sum of the argument and 1.
BIND (ofn:log1p(4) AS ?log1p)

# max
# The greater of two numbers.
BIND (ofn:max(3, 5) AS ?max)

# min
# The smaller of two numbers.
BIND (ofn:min(3, 5) AS ?min)

# nextAfter
# Returns the floating-point number adjacent to the first argument in the direction of the second_
↪ argument.
BIND (ofn:nextAfter(2, -7) AS ?nextAfter)

# nextDown
# Returns the floating-point value adjacent to d in the direction of negative infinity.
BIND (ofn:nextDown(2) AS ?nextDown)

# nextUp
# Returns the floating-point value adjacent to d in the direction of positive infinity.
BIND (ofn:nextUp(2) AS ?nextUp)

# pi
# The Pi constant.
BIND (ofn:pi() AS ?pi)

# pow
# The power function.
BIND (ofn:pow(2, 3) AS ?pow)

```



```

# rint
# Returns the double value that is closest in value to the argument and is equal to a mathematical
↪integer.
BIND (ofn:rint(2.51) AS ?rint)

# scalb
# Returns  $x \times 2^{\text{scaleFactor}}$  rounded as if performed by a single correctly rounded floating-point
↪multiply
# to a member of the double value set.
BIND (ofn:scalb(3, 3) AS ?scalb)

# signum
# 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.
BIND (ofn:signum(-5) AS ?signum)

# sin
# The sine function. The argument is in radians.
BIND (ofn:sin(2) AS ?sin)

# sinh
# The hyperbolic sine function.
BIND (ofn:sinh(2) AS ?sinh)

# sqrt
# The square root function.
BIND (ofn:sqrt(2) AS ?sqrt)

# tan
# The tangent function. The argument is in radians.
BIND (ofn:tan(1) AS ?tan)

# tanh
# The hyperbolic tangent function.
BIND (ofn:tanh(1) AS ?tanh)

# toDegrees
# Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
BIND (ofn:toDegrees(1) AS ?toDegrees)

# toRadians
# Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
BIND (ofn:toRadians(1) AS ?toRadians)

# ulp
# 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  $x$ ,  $\text{ulp}(-x) == \text{ulp}(x)$ . See
↪Math.ulp().
BIND (ofn:ulp(1) AS ?ulp)
}

```

Note: All function arguments should be bound.

The result of the query evaluation is:

```
acos="1.0471975511965979"^^<http://www.w3.org/2001/XMLSchema#double>
asin="0.5235987755982989"^^<http://www.w3.org/2001/XMLSchema#double>
atan="0.7853981633974483"^^<http://www.w3.org/2001/XMLSchema#double>
atan2="1.5707963267948966"^^<http://www.w3.org/2001/XMLSchema#double>
cbrt="1.2599210498948732"^^<http://www.w3.org/2001/XMLSchema#double>
copySign="-2.0"^^<http://www.w3.org/2001/XMLSchema#double>
cos="0.5403023058681398"^^<http://www.w3.org/2001/XMLSchema#double>
cosh="1.543080634815244"^^<http://www.w3.org/2001/XMLSchema#double>
e="2.718281828459045"^^<http://www.w3.org/2001/XMLSchema#double>
exp="7.38905609893065"^^<http://www.w3.org/2001/XMLSchema#double>
expm1="19.085536923187668"^^<http://www.w3.org/2001/XMLSchema#double>
floorDiv="2.0"^^<http://www.w3.org/2001/XMLSchema#double>
floorMod="1.0"^^<http://www.w3.org/2001/XMLSchema#double>
getExponent="3.0"^^<http://www.w3.org/2001/XMLSchema#double>
hypot="5.0"^^<http://www.w3.org/2001/XMLSchema#double>
IEEEremainder="-1.0"^^<http://www.w3.org/2001/XMLSchema#double>
log10="0.6020599913279624"^^<http://www.w3.org/2001/XMLSchema#double>
log="1.3862943611198906"^^<http://www.w3.org/2001/XMLSchema#double>
log1p="1.6094379124341003"^^<http://www.w3.org/2001/XMLSchema#double>
max="5.0"^^<http://www.w3.org/2001/XMLSchema#double>
min="3.0"^^<http://www.w3.org/2001/XMLSchema#double>
nextAfter="1.9999999999999998"^^<http://www.w3.org/2001/XMLSchema#double>
nextDown="1.9999999999999998"^^<http://www.w3.org/2001/XMLSchema#double>
nextUp="2.0000000000000004"^^<http://www.w3.org/2001/XMLSchema#double>
pi="3.141592653589793"^^<http://www.w3.org/2001/XMLSchema#double>
pow="8.0"^^<http://www.w3.org/2001/XMLSchema#double>
rint="3.0"^^<http://www.w3.org/2001/XMLSchema#double>
scalb="24.0"^^<http://www.w3.org/2001/XMLSchema#double>
signum="-1.0"^^<http://www.w3.org/2001/XMLSchema#double>
sin="0.9092974268256817"^^<http://www.w3.org/2001/XMLSchema#double>
sinh="3.626860407847019"^^<http://www.w3.org/2001/XMLSchema#double>
sqrt="1.4142135623730951"^^<http://www.w3.org/2001/XMLSchema#double>
tan="1.5574077246549023"^^<http://www.w3.org/2001/XMLSchema#double>
tanh="0.7615941559557649"^^<http://www.w3.org/2001/XMLSchema#double>
toDegrees="57.29577951308232"^^<http://www.w3.org/2001/XMLSchema#double>
toRadians="0.017453292519943295"^^<http://www.w3.org/2001/XMLSchema#double>
ulp="2.220446049250313E-16"^^<http://www.w3.org/2001/XMLSchema#double>
```

8.8 OWL compliance

GraphDB supports several OWL like dialects: OWL Horst (owl-horst), OWL Max (owl-max), which covers most of OWL Lite and RDFS, OWL2 QL (owl2-ql) and OWL2 RL (owl2-rl).

With the owl-max ruleset, GraphDB supports the following semantics:

- full RDFS semantics without constraints or limitations, apart from the entailment related to typed literals (known as D-entailment). For instance, meta-classes (and any arbitrary mixture of class, property, and individual) can be combined with the supported OWL semantics;
- most of OWL Lite;
- all of OWL DLP.

The differences between OWL Horst and the OWL dialects supported by GraphDB (owl-horst and owl-max) can be summarised as follows:

- GraphDB does not provide the extended support for typed literals, introduced with the D-entailment extension of the RDFS semantics. Although such support is conceptually clear and easy to implement, it is our understanding that the performance penalty is too high for most applications. You can easily implement the rules defined for this purpose by ter Horst and add them to a custom ruleset;
- There are no inconsistency rules by default;

- A few more OWL primitives are supported by GraphDB (ruleset owl-max);
- There is extended support for schema-level (T-Box) reasoning in GraphDB.

Even though the concrete rules pre-defined in GraphDB differ from those defined in OWL Horst, the complexity and decidability results reported for R-entailment are relevant for TRREE and GraphDB. To be more precise, the rules in the owl-horst ruleset do not introduce new B-Nodes, which means that R-entailment with respect to them takes polynomial time. In KR terms, this means that the owl-horst inference within GraphDB is tractable.

Inference using owl-horst is of a lesser complexity compared to other formalisms that combine DL formalisms with rules. In addition, it puts no constraints with respect to meta-modelling.

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.

8.9 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.

Materialisation 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 modelling 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.

RELEASE NOTES

What's in this document?

- *GraphDB 9.0.0*
 - *Component versions*
 - *GraphDB Engine*
 - *GraphDB Workbench*
 - *GraphDB Plugins*
 - *GraphDB Connectors*
 - *GraphDB Cluster*

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., 8.1.2 where the major version is 8, the minor version is 1 and the patch version is 2.

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 8 includes two components with their version numbers:

- RDF4J
- GraphDB Connectors

Their versions use the same semantic versioning scheme as the whole product, and their values are provided only as a reference.

9.1 GraphDB 9.0.0

Released: 30 September 2019

9.1.1 Component versions

RDF4J	Connectors
3.0.0	12.0.0

Important:

- The new release features a major redesign of the database Plugin API that simplifies the way to implement complex software logic next to the core database engine. Ontotext releases multiple plugins to the community as open source, demonstrating how to solve common tasks such as data virtualization (see [MongoDB Plugin](#)), a complex ranking of search results (see [Autocomplete Plugin](#)), new types of indexing (see [GeoSPARQL plugin](#)) and advanced graph analytics (see [RDFRank plugin](#)).
 - [GraphDB Workbench](#) now becomes a separate open source project enabling the fast development of knowledge graph prototypes starting from the default AngularJS administrative interface. Furthermore, the product includes an open source [GraphDB.js](#) driver optimized for Node.js and other rapid development frameworks.
 - GraphDB 9.0 supports OpenJDK 11. The product continues to support OpenJDK 8 for its existing clients, as well as every new Oracle Java Long Term Support (LTS) version.
 - The product continues to support the RDF open source community, and is now upgraded to [RDF4J version 3.0](#).
-

9.1.2 GraphDB Engine

9.1.2.1 New features

- GDB-3607 Move to OpenJDK and support for newer Java versions. Make OpenJDK 8 and 11 the officially tested and recommended Java versions
- GDB-3649 Upgrade to RDF4J 3.0

9.1.2.2 Bug fixing

- GDB-1416 Exceptions in query evaluation do not result in error messages
- GDB-3494 GraphDB is trying to rollback non-existing transaction
- GDB-3506 Concurrent requests for storage health checks may produce an error
- GDB-3531 Exception executing simple query after PUT statements in a repo
- GDB-3585 Suboptimal execution plan for construct with FILTER
- GDB-3592 Deadlock when notification listeners are to be notified but they were garbage collected before that
- GDB-3596 LiteralIndex failures within RepositoryRule
- GDB-3643 Join of subqueries does not work correctly
- GDB-3646 Error “Could not convert the query to our optimized model” reported during query evaluation containing MINUS
- GDB-3703 Graph Replacement Optimization does not work with the Java API
- GDB-3720 Compound transactions do not reflect change in size()
- GDB-3830 OPTIONAL and Alt-prop-path causes mix-up of bindings
- GDB-3843 Bulk update “getRealId() must be called in exclusive transaction or after precommit()”

9.1.3 GraphDB Workbench

9.1.3.1 New features

- GDB-3619 Split Workbench code base from GraphDB and open source the UI code
- GDB-3728 Extend Cypress tests to cover all functionalities of the Workbench

9.1.3.2 Bug fixing

- GDB-3248 Visual graph: expanding nodes “over owl:sameAs” not working properly
- GDB-3642 Visual graph UI missing buttons

9.1.4 GraphDB Plugins

9.1.4.1 New features

- GDB-1801 Open source plugin API and plugins with developer examples
- GDB-3308 Implement a way for plugins to know if they are in a cluster
- GDB-3631 Plugin API: Redesign the plugin notification interfaces to handle the notifications about transactions
- GDB-3633 Plugin API: Extend system info object passed at plugin initialization
- GDB-3635 Plugin API: Add mechanism to access the fingerprint in plugins
- GDB-3636 Plugin API: Cleanup unused parts of the Plugin API like GlobalViewOnData and MemoryConfigurable
- GDB-3730 Update Plugin API, plugin location, and the Javadoc
- GDB-3803 Open source the GeoSPARQL plugin
- GDB-3806 Open source the RDF Rank plugin
- GDB-3807 Open source the Autocomplete plugin
- GDB-3808 Open source the Lucene FTS plugin
- GDB-3813 Open source the GeoSpatial plugin
- GDB-3814 Open source the Notifications Logger plugin
- GDB-3815 Open source the MongoDB plugin

9.1.4.2 Bug fixing

- GDB-3309 Asynchronous RDFRank build through SPARQL in cluster breaks the cluster
- GDB-3453 Explain the slow performance for rebuilding GeoSPARQL on Kadaster repository
- GDB-3511 GeoSPARQL: Creating a quad plugin with precision >24 will result in an error when trying to create a geohash plugin
- GDB-3554 Similarity indices with training cycles use vector values from first iteration on search
- GDB-3556 Cannot build predication similarity index over PubMed authors data
- GDB-3584 Similarity text index -porterstemmer flag does not work
- GDB-3599 Stopwords do not work with default analyzer in text similarity indices

9.1.5 GraphDB Connectors

9.1.5.1 New features

- GDB-3582 Upgrade connectors to latest major releases of Lucene/Solr (8.x) and Elasticsearch (7.x)

9.1.5.2 Bug fixing

- GDB-3816 Wrong exporting of 'elasticsearchClusterSniff' setting from Connectors view
- GDB-3819 Viewing existing connectors does not show the values of options that were set by default value and not explicitly
- GDB-3905 Repair does not work when repairing a connector marked as “incompatible” by being created by an older version

9.1.6 GraphDB Cluster

9.1.6.1 Bug fixing

- GDB-3766 Cluster incorrectly indicates IsWritable=false when one of its workers is down

Where does the name “OWLIM” (the former GraphDB name) come from? The name 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.

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*.

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](#).

Is GraphDB Jena-compatible? Yes, GraphDB is compatible with [Jena 2.7.3](#) with a built-in adapter. | For more information, see [using-graphdb-with-jena](#)

What are the advantages of using solid-state drives as opposed to hard-disk drives? 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 indices. Despite utilising 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.

How to find out the exact version number of GraphDB? The major/minor version and patch number are part of the GraphDB distribution .zip file name. They can also be seen at the bottom of the GraphDB Workbench home page, together with the RDF4J, Connectors, and Plugin API's versions.

A second option is to run the `graphdb -v` startup script command if you are running *GraphDB as a standalone server* (without Workbench). It will also return the build number of the distribution.

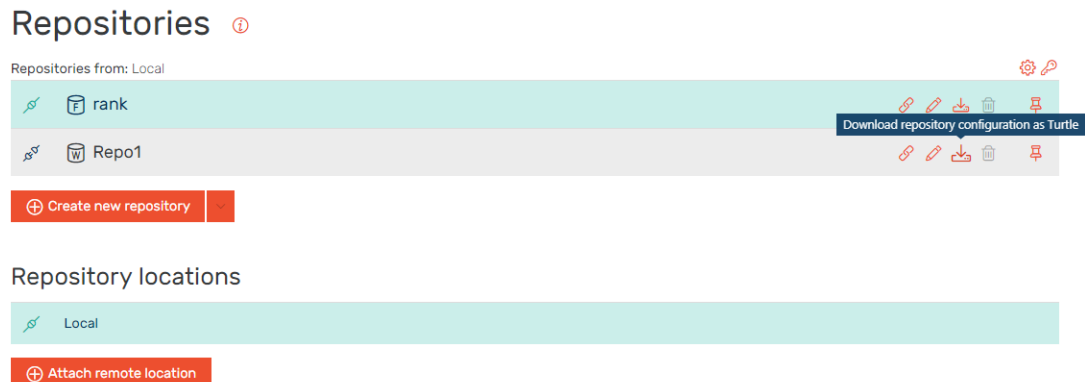
Another option is to run the following DESCRIBE query in the Workbench SPARQL editor:

```
DESCRIBE <http://www.ontotext.com/SYSINFO> FROM <http://www.ontotext.com/SYSINFO>
```

It returns pseudo-triples providing information on various GraphDB states, including the number of triples (total and explicit), storage space (used and free), commits (total and whether there are any active ones), the repository signature, and the build number of the software.

How to retrieve repository configurations?

To see what configuration data is stored in a GraphDB repository, go to *Repositories* and use *Download repository configuration as Turtle* icon



Then you could open the result file named repositoryname-config.ttl

Why can't I use my 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).

How to speed up a slow repository with enabled security when each request includes HTTP basic authentication?

Every HTTP authentication request takes significant time due to the bcrypt algorithm which encrypts the clear text password to match with the hash in \$GDB_HOME/work/workbench/settings.js.

The best solution is to do one-time authentication and keep the JWT string. JWT has a much lower overhead.

1. Login and generate the authorization

```
curl -X POST -I 'http://localhost:7200/rest/login/admin' -H 'X-GraphDB-Password: root'
```

2. Pass the returned JWT key

```
curl -H 'Authorization: GDB_eyJ1c2VybWVtZSI6ImFkbWluIiwiaXV0aGVudGljYXRlZEF0IjoNTU30TIzNTkxNDA0fQ==.OwSkajbUoHHSQGfwaCxbb1f7bn0PJUeL4VbGEmNcWY=' http://localhost:7200/repositories/SYSTEM/size
```

The JWT token expires every 30 days.

To change the encryption algorithm from bcrypt to SHA-256 supported by the older GDB version, update the password token in \$GDB_HOME/work/workbench/settings.js with the encrypted value of: password{user}

```
echo root{admin} | sha256sum
```

```
{
  "admin": {
    "username": "admin",
    "password": "9ca992309ea8a83c7e3af185e645f7d56dce169310a65f67d4708f9b023fb297",
    "grantedAuthorities": [
      "READ_REPO_*",
      "ROLE_USER"
    ],
  },
  "appSettings": {
    "DEFAULT_SAMEAS": true,
    "DEFAULT_INFERENCE": true,
    "EXECUTE_COUNT": true,
    "IGNORE_SHARED_QUERIES": false
  },
  "dateCreated": 1557904143163
}
```

SUPPORT

- email: graphdb-support@ontotext.com
- Twitter: [@OntotextGraphDB](https://twitter.com/OntotextGraphDB)
- GraphDB tag on Stack Overflow at <http://stackoverflow.com/questions/tagged/graphdb>