Stardog 3: The Manual

Stardog 3.1.4 (28 July 2015)

Introduction

Enterprise Premium Support

Quick Start Guide

Using Stardog

Administering Stardog

Security

High Availability Cluster

OWL & Rule Reasoning

Validating Constraints

Programming Stardog

Java Programming

Network Programming

Javascript Programming

Clojure Programming

.Net Programming

Spring Programming

Groovy Programming

SNARL Migration Guide

Understanding Stardog

Appendix

Stardog is a fast, semantic graph database for the enterprise. Check out the Quick Start Guide to get Stardog installed and running in five easy steps.

What’s New

In 3.1...

- √ Named Graph Security
- √ Cluster Management
- √ Cluster performance improvements
- √ Web Console improvements

In 3.0...

- √ High Availability Cluster
- √ Enterprise Premium Support
- √ Improved query performance
- √ Equality reasoning
- √ Improved incremental write performance
- √ ICV repair plans
- √ User-defined search indexers
- √ User-defined aggregate functions
- √ Maven support
- √ Enterprise authentication via LDAP
- √ Simplified reasoning levels

Introduction

Stardog is a semantic graph database, available in client-server, middleware, and embedded modes. Stardog **3.1.4** (**28 July 2015**) supports the RDF graph data model; SPARQL query language; HTTP and SNARL protocols for remote access and control; OWL 2 and user-defined rules for inference and data analytics; and programmatic interaction via several languages and network interfaces. Stardog is made with skill, taste, and a point of view in DC, Boston, and Germany.

To learn more about where we've been and where we're headed, consult the RELEASE NOTES and milestones.

Stardog Version	HTML	PDF
3.1.4	√	√
3.1.3	√	√
3.1.2	√	√
3.1.1	√	√
3.1	√	√
3.0.2	√	√
3.0.1	√	√
3.0	√	√

Downloading Stardog & Support

Download Stardog to get started. The Stardog support forum is the place to report bugs, ask questions, etc. Stardog developers may be found on Gitter chat, too.

Contributing

There are several open source components of Stardog; feel free to submit pull requests: stardog-docs, stardog.js, stardog-groovy, stardog-spring, stardog.rb, and stardog-clj. Many thanks to Stardog customers, users, contributors, testers, etc. You know who you are, and so do we. Thank you![1] You can also help by editing these docs on Github if you spot a problem.

Roadmap

Stardog 4.0

- Virtual Graphs via R2RML
- Property Graph/Tinker Pop 3 support
- Geospatial query answering
- Remove dependency on Cluster proxy
- **move to Java** 8
- Pelorus integrated with Stardog Web Console

Stardog 4.x

- native HDFS support
- Distributed transaction performance improvements
- Stardog Constellation, a unified graph computation framework
- Virtual enterprise graphs from HDFS-based data lakes

Enterprise Premium Support

Customers of Stardog’s Enterprise Premium Support have access to the following capabilities and services. Enterprise Premium Support is available as of Stardog 3 release. Email for details.

Real-time Support

Get access to the core Stardog development team in real-time via voice or chat. Let us help you get the most from Stardog, 24/7. Our core team has more semantic graph application and tool development experience than any other team on the planet. Other vendors shunt you off to inexperienced level one techs. We’ve never done that and never will.

Private Maven Repositories

See Using Maven for details; this includes a per-customer, private Maven repository, CDN-powered, for 24/7 builds, updates, and feature releases.

We’re also tying Maven and Docker together, providing private Docker repositories for customers, which allows us to build out clusters, custom configurations, best practices, and devops tips-and-tricks into custom Docker images…so that you don’t have to.

Private Docker Repositories

Docker-based deliverables not only shortens your development and devops cycles but keeps you up-to-date with the latest-greatest versions of Stardog, including security fixes, performance hot fixes, and deployment best practices and configurations.

Some day all software will be distributed via containers; the days of the monolithic release, held for weeks or month to suit business cycles, are over. You heard it here first; well, okay, not **first** but you **got** it here **early**.

Priority Bug Fixes

With Maven and Docker in place, we’ve got a software delivery mechanism ready to push priority bug fixes into your enterprise as soon as they’re ready. We’ve averaged one Stardog release every two weeks since 2012. Enterprise Premium Support customers can now take advantage of our development pace in a controlled fashion.

Priority Feature Releases

We hate holding new features in a feature branch, especially for mundane business reasons; we want to release new stuff as soon as possible to our customers. With Enterprise Premium Support, we can maintain a disruptive pace of innovation without disrupting **you**.

Quick Start Guide

Requirements

It just doesn't get any easier than this: Stardog runs on Java 6, 7 and 8. Stardog runs best on, but does not require, a 64-bit JVM that supports `sun.misc.Unsafe`.

Insecurity

We optimize Stardog out-of-the-box for ease and simplicity. You should take additional steps to secure it before production deployement. It's easy and it's smart, so just do it. In case that's not blunt enough:

> **NOTE**
>
> Stardog ships with an **insecure** but usable default setting: the super user is `admin` and the `admin` password is "admin". This is fine until it isn't, at which point you should read the Security section.

Linux and OSX

1. Tell Stardog where its home directory (where databases and other files will be stored) is

   ```
   $ export STARDOG_HOME=/data/stardog
   ```

If you’re using some weird Unix shell that doesn’t create environment variables in this way, adjust accordingly. **If `STARDOG_HOME` isn’t defined, Stardog will use the Java `user.dir` property value.**

> **NOTE**
>
> You should consider the upgrade process when setting `STARDOG_HOME` for production or other serious usage. In particular, you probably don’t want to set the directory where you install Stardog as `STARDOG_HOME` as that makes upgrading less easy. Set `STARDOG_HOME` to some other location.

2. Copy the `stardog-license-key.bin` into the right place:

   ```
   $ cp stardog-license-key.bin $STARDOG_HOME
   ```

   Of course `stardog-license-key.bin` has to be readable by the Stardog process.

   Stardog won’t run without a valid `stardog-license-key.bin` in `STARDOG_HOME`.

3. Start the Stardog server. By default the server will expose SNARL and HTTP interfaces on port 5820.[2]

   ```
   $ ./stardog-admin server start
   ```

4. Create a database with an input file:

   ```
   $ ./stardog-admin db create -n myDB examples/data/University0_0.owl
   ```

5. Query the database:

   ```
   $ ./stardog query myDB "SELECT DISTINCT ?s WHERE { ?s ?p ?o } LIMIT 10"
   ```

   You can use the Web Console to search or query the new database you created by visiting http://localhost:5820/myDB in your browser.

Now, go have a drink: you’ve earned it.

Windows

Windows…really? Okay, but don't blame us if this hurts…The following steps are carried out using the Windows command prompt which you can find under Start ‣ Programs ‣ Accessories ‣ Command Prompts or Start ‣ Run ‣ `cmd`.

First, tell Stardog where its home directory (where databases and other files will be stored) is:

```
> set STARDOG_HOME=C:\data\stardog
```

Second, copy the `stardog-license-key.bin` into the right place:

```
> COPY /B stardog-license-key.bin %STARDOG_HOME%
```

The `/B` is required to perform a binary copy or the license file may get corrupted. Of course `stardog-license-key.bin` has to be readable by the Stardog process. Finally, Stardog won't run without a valid `stardog-license-key.bin` in `STARDOG_HOME`.

Third, start the Stardog server. By default the server will expose SNARL and HTTP interfaces on port 5820.[3]

```
> stardog-admin.bat server start
```

This will start the server in the current command prompt, you should leave this window open and open a new command prompt window to continue.

Fourth, create a database with an input file:

```
> stardog-admin.bat db create -n myDB examples/data/University0_0.owl
```

Fifth, query the database:

```
> stardog.bat query myDB "SELECT DISTINCT ?s WHERE { ?s ?p ?o } LIMIT 10"
```

You can use the Web Console to search or query the new database you created by hitting http://localhost:5820/myDB in your browser.

You should drink the whole bottle, brave Windows user!

Using Stardog

Stardog supports SPARQL, the W3C standard for querying RDF graphs, as well as a range of other capabilities, including updating, versioning exporting, searching, obfuscating, and browsing graph data.

Querying

Stardog supports SPARQL 1.1 and also the OWL 2 Direct Semantics entailment regime.

To execute a SPARQL query against a Stardog database, use the `query` subcommand:

```
$ stardog query myDb "select * where { ?s ?p ?o }"
```

Detailed information on using the query command in Stardog can be found on its `man` page.

DESCRIBE

SPARQL's `DESCRIBE` keyword is deliberately underspecified; vendors are free to do, for good or bad, whatever they want. In Stardog a `DESCRIBE <theResource>` query retrieves the predicates and objects for all the triples for which `<theResource>` is the subject. There are, of course, about seventeen thousand other ways to implement `DESCRIBE`; we've implemented four or five of them and may expose them to users in a future release of Stardog *based on user feedback and requests*.

Query Functions

Stardog supports all of the functions in SPARQL, as well as some others from XPath and SWRL. Any of these functions can be used in queries or rules. Some functions appear in multiple namespaces, but all of the namespaces will work.

See SPARQL Query Functions for the complete list.

Federated Queries

Stardog 3.0 adds support for the SERVICE keyword[4] which allows users to query distributed RDF via SPARQL-compliant data sources. You can use this to federate queries between several Stardog database or Stardog and other public endpoints.

> **NOTE** Stardog 3.0 doesn't support variables for the service URL (`SERVICE ?serviceURL`).

Stardog ships with a default `Service` implementation which uses SPARQL Protocol to send the service fragment to the remote endpoint and retrieve the results. Any endpoint that conforms to the SPARQL protocol can be used.

The Stardog SPARQL endpoint is `http://<server>:<port>/{db}/query` .

HTTP Authentication

Stardog requires authentication. If the endpoint you're referencing with the `SERVICE` keyword requires HTTP authentication, credentials are stored in a password file called `services.sdpass` located in `STARDOG_HOME` directory. The default `Service` implementation assumes HTTP BASIC authentication; for services that use DIGEST auth, or a different authentication mechanism altogether, you'll need to implement a custom `Service` implementation.

Updating

There are many ways to update the data in a Stardog database; two of the most commonly used methods are the CLI and SPARQL Update queries, each of which are discussed below.

SPARQL Update

SPARQL 1.1 Update can be used to insert RDF into or delete RDF from a Stardog database using SPARQL query forms `INSERT` and `DELETE` , respectively.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
INSERT DATA
{ GRAPH <http://example/bookStore> { <http://example/book1>  ns:price  42 } }
```

An example of deleting RDF:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

DELETE DATA
{
  <http://example/book2> dc:title "David Copperfield" ;
                         dc:creator "Edmund Wells" .
}
```

Or they can be combined with `WHERE` clauses:

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>

WITH <http://example/addresses>
DELETE { ?person foaf:givenName 'Bill' }
INSERT { ?person foaf:givenName 'William' }
WHERE
  { ?person foaf:givenName 'Bill' }
```

NOTE

Per the SPARQL Update spec, Stardog treats Update queries as implicitly transactional and atomic. Since Stardog does not support nested

> transactions, it will not (currently) support an Update query in an open transaction.[5]

Adding Data with the CLI

As of Stardog 3.1.4, the most efficient way to load data into Stardog is at database creation time. See the Creating a Database section for bulk loading data at database creation time. To add data to an existing Stardog database, use the `add` command:

```
$ stardog data add myDatabase 1.rdf 2.rdf 3.rdf
```

The optional arguments are `-f` (or `--format`) to specify the RDF serialization type of the files to be loaded; if you specify the wrong type, `add` will fail. If you don’t specify a type, Stardog will try to determine the type on its own based on the file extension. For example, the files that have names ending with '.ttl' will be parsed with Turtle syntax. If you specify a type, then all the files being loaded must of that same type.

If you want to add data to a named graph, specify it via the `--named-graph` or `-g` options.

Removing Data with the CLI

To remove data, use `remove` with the following input(s):

1. one Named Graph, *or*
2. one or more files containing RDF (in some recognized serialization format, i.e., RDF/XML, Turtle, Trig), *or*
3. one Named Graph and one or more RDF files.

For example,

```
$ stardog data remove -g http://foo myDatabase
```

will remove the named graph `http://foo` and all its triples from `myDatabase`.

```
$ stardog data remove myDatabase 1.rdf
```

will remove the triples in `1.rdf` from (the default graph of) `myDatabase`.

```
$ stardog data remove -g http://foo -f TURTLE myDatabase 2.rdf 3.rdf
```

will remove the triples in the Turtle files `2.rdf` and `3.rdf` from the named graph `http://foo` of `myDatabase`.

How Stardog Handles RDF Parsing

RDF parsing in Stardog is strict: it requires typed RDF literals to match their explicit datatypes, URIs to be well-formed, etc. In some cases, strict parsing isn't ideal—it may be disabled using the `strict.parsing` configuration option.

However, even with strict parsing disabled, Stardog's RDF parser may encounter parse errors from which it cannot recover. And loading data in lax mode may lead to unexpected SPARQL query results. For example, malformed literals (`"2.5"^^xsd:int`) used in filter evaluation may lead to undesired results.

Versioning

Stardog supports graph change management capability that lets users track changes between revisions of a Stardog database, add comments and other metadata to the revisions, extract diffs between those revisions, tag revisions with labels, and query over the revision history of the database using SPARQL.

Versioning support for a database is disabled by default but can be enabled at any time by setting the configuration option `versioning.enabled` to true. For example, you can create a database with versioning support as follows:

```
$ stardog-admin db create -o versioning.enabled=true -n myDb
```

This option can also be set after database creation using the `stardog-admin metadata set` command.

The following examples give a **very brief overview** of this capability; see the Man Pages for all the details.

Committing Changes

Commit a new version by adding and removing triples specified in files. Different from the `data add/remove` commands, `commit` allows one to add and remove triples in one commit and to associate a commit message.

NOTE Removals are performed before additions.

To commit changes:

```
$ stardog vcs commit --add add_file1.ttl add_file2.ttl --remove remove_file.ttl -m
"This is an example commit" myDb
```

Viewing Revisions

To see all revisions (commits) in a database:

```
$ stardog vcs list myDb
$ stardog vcs list --committer userName myDb
```

The output can be tweaked using `--after`, `--before`, and `--committer`.

Reverting Revisions

You can revert specific revisions, ranges, etc.

```
$ stardog vcs revert myDb
$ stardog vcs revert myDb de44369d-cc7b-4244-a3fb-3f6e271420b0
```

Viewing Diffs

You can also see the differences between revisions; by default, between the head version and its previous versions or the changes in a specific commit, respectively:

```
$ stardog vcs diff myDb
$ stardog vcs diff myDb de44369d-cc7b-4244-a3fb-3f6e271420b0
```

NOTE Diffs are represented as SPARQL Update queries so that they may be used as a kind of graph patch.

Using Tags

You can also create, drop, list tags, i.e., named revisions:

```
$ stardog vcs tag --list myDb
```

Querying the Revision History

The revision history of the database is represented as RDF using the W3C PROV vocabulary and can be queried using SPARQL:[6]

```
$ stardog vcs query myDb 'SELECT...'
```

Exporting

To export data from a Stardog database back to RDF, `export` is used by specifying

1. the name of the database to export
2. optionally, the URIs of the named graphs to export if you wish to export specific named graphs only. Keywords `DEFAULT` and `ALL` can be used as values of the `--named-graph` parameter to export the default graph and all graphs, respectively

3. the export format: `N-TRIPLES, RDF/XML, TURTLE, TRIG`. The default is `N-TRIPLES`. `TRIG` must be used when exporting the entire database if the database contains triples inside named graphs
4. a file to export to

For example,

```
$ stardog data export --format TURTLE myDatabase myDatabase_output.ttl

$ stardog data export --named-graph http://example.org/context myDatabase
myDatabase_output.nt
```

Searching

Stardog includes an RDF-aware semantic search capability: it will index RDF literals and supports information retrieval-style queries over indexed data. See Managing Search for more details.

Indexing Strategy

The indexing strategy creates a "search document" per RDF literal. Each document consists of the following fields: literal ID; literal value; and contexts.

See User-defined Lucene Analyzer for details on customizing Stardog’s search programmatically.

Search in SPARQL

We use the predicate `http://jena.hpl.hp.com/ARQ/property#textMatch` to access the search index in a SPARQL query.

The `textMatch` function has one required argument, the search query in Lucene syntax. It also has two optional arguments. The first is the score threshold; results whose score is **below** this number are not included in the results. The second is the maximum number of results to return from Lucene. When not specified, the default limit is `50`. When a `LIMIT` is specified in the

SPARQL query, it does not affect the full-text search, that only restricts the size of the final result set.

For example,

```
SELECT DISTINCT ?s ?score
WHERE {
?s ?p ?l.
( ?l ?score ) <http://jena.hpl.hp.com/ARQ/property#textMatch> ( 'mac' 0.5 50 ).
}
```

This query selects the top 50 literals, and their scores, which match 'mac' and whose scores are above a threshold of 0.5. These literals are then joined with the generic BGP `?s ?p ?l` to get the resources (?s) that have those literals. Alternatively, you could use `?s rdf:type ex:Book` if you only wanted to select the books which reference the search criteria; you can include as many other BGPs as you like to enhance your initial search results.

Escaping Characters in Search

The "/" character must be escaped because Lucene says so. In fact, there are several characters that are part of Lucene’s query syntax that must be escaped.

Searching with the Command Line

First, check out the `search` man page:

```
$ stardog help query search
```

Okay, now let’s do a search over the O’Reilly book catalog in RDF for everything mentioning "html":

```
$ stardog query search -q "html" -l 10 catalog
```

The results?

```
Index    Score    Hit
====================
0    6.422    urn:x-domain:oreilly.com:product:9780596527402.IP
1    6.422    urn:x-domain:oreilly.com:product:9780596003166.IP
2    6.422    urn:x-domain:oreilly.com:product:9781565924949.IP
3    6.422    urn:x-domain:oreilly.com:product:9780596002251.IP
4    6.422    urn:x-domain:oreilly.com:product:9780596101978.IP
5    6.422    urn:x-domain:oreilly.com:product:9780596154066.IP
6    6.422    urn:x-domain:oreilly.com:product:9780596157616.IP
7    6.422    urn:x-domain:oreilly.com:product:9780596805876.IP
8    6.422    urn:x-domain:oreilly.com:product:9780596527273.IP
9    6.422    urn:x-domain:oreilly.com:product:9780596002961.IP
```

Query Syntax

Stardog search is based on Lucene 4.2.0: we support all of the search modifiers that Lucene supports, with the exception of fields.

- wildcards: `?` and `*`
- fuzzy: `~` and `~` with similarity weights (e.g. `foo~0.8`)
- proximities: `"semantic web"~5`
- term boosting
- booleans: `OR`, `AND`, `NOT`, ``+ , and `-``.
- grouping

For a more detailed discussion, see the Lucene docs.

Obfuscating

When sharing sensitive RDF data with others, you might want to (selectively) obfuscate it so that sensitive bits are not present, but non-sensitive bits remain. For example, this feature can be used to submit Stardog bug reports using sensitive data.

Data obfuscation works much the same way as the `export` command and supports the same set of arguments:

```
$ stardog data obfuscate myDatabase obfDatabase.ttl
```

By default, all URIs, bnodes, and string literals in the database will be obfuscated using the SHA256 message digest algorithm. Non-string typed literals (numbers, dates, etc.) are left unchanged as well as URIs from built-in namespaces (`RDF` , `RDFS` , and `OWL`). It's possible to customize obfuscation by providing a configuration file.

```
$ stardog data obfuscate --config obfConfig.ttl myDatabase  obfDatabase.ttl
```

The configuration specifies which URIs and strings will be obfuscated by defining inclusion and exclusion filters. See the example configuration file provided in the distribution for details.

Once the data is obfuscated, queries written against the original data will no longer work. Stardog provides query obfuscation capability, too, so that queries can be executed against the obfuscated data. If a custom configuration file is used to obfuscate the data, then the same configuration should be used for obfuscating the queries as well:

```
$ stardog query obfuscate --config obfConfig.ttl myDatabase myQuery.sparql >
obfQuery.ttl
```

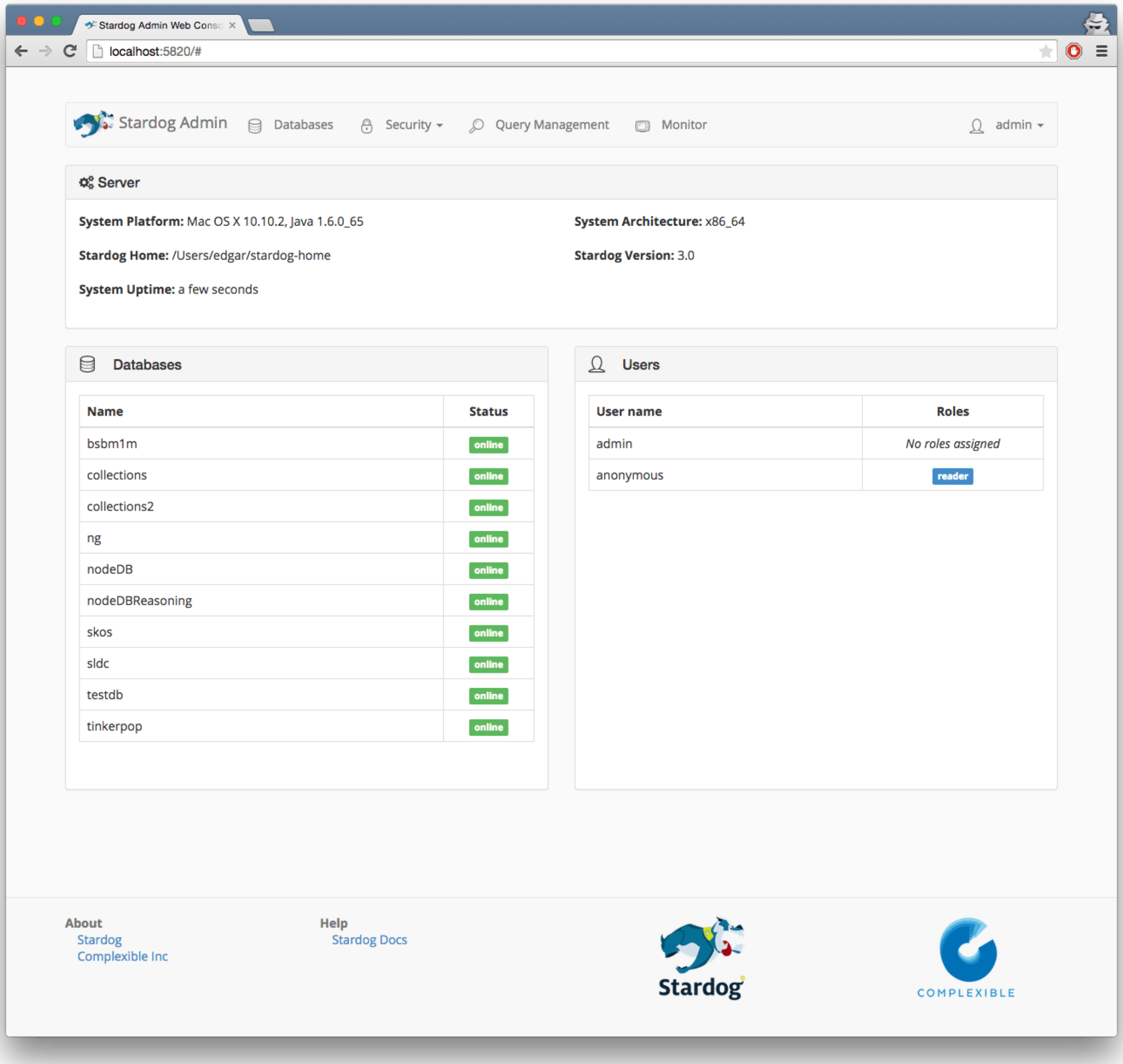
Browsing

The Stardog Web Console is a responsive web app for the Stardog Server and for every Stardog database that makes administration and interaction with data quick and easy; you can access it at `http://foo:5820` where `foo` is the name of the machine where Stardog is running.

A Screenshot Tour…

Seriously, this is a lot more fun if you just download the damn thing and hit it with a browser!

Web Console



Server

System Platform: Mac OS X 10.10.2, Java 1.6.0_65

System Architecture: x86_64

Stardog Home: /Users/edgar/stardog-home

Stardog Version: 3.0

System Uptime: a few seconds

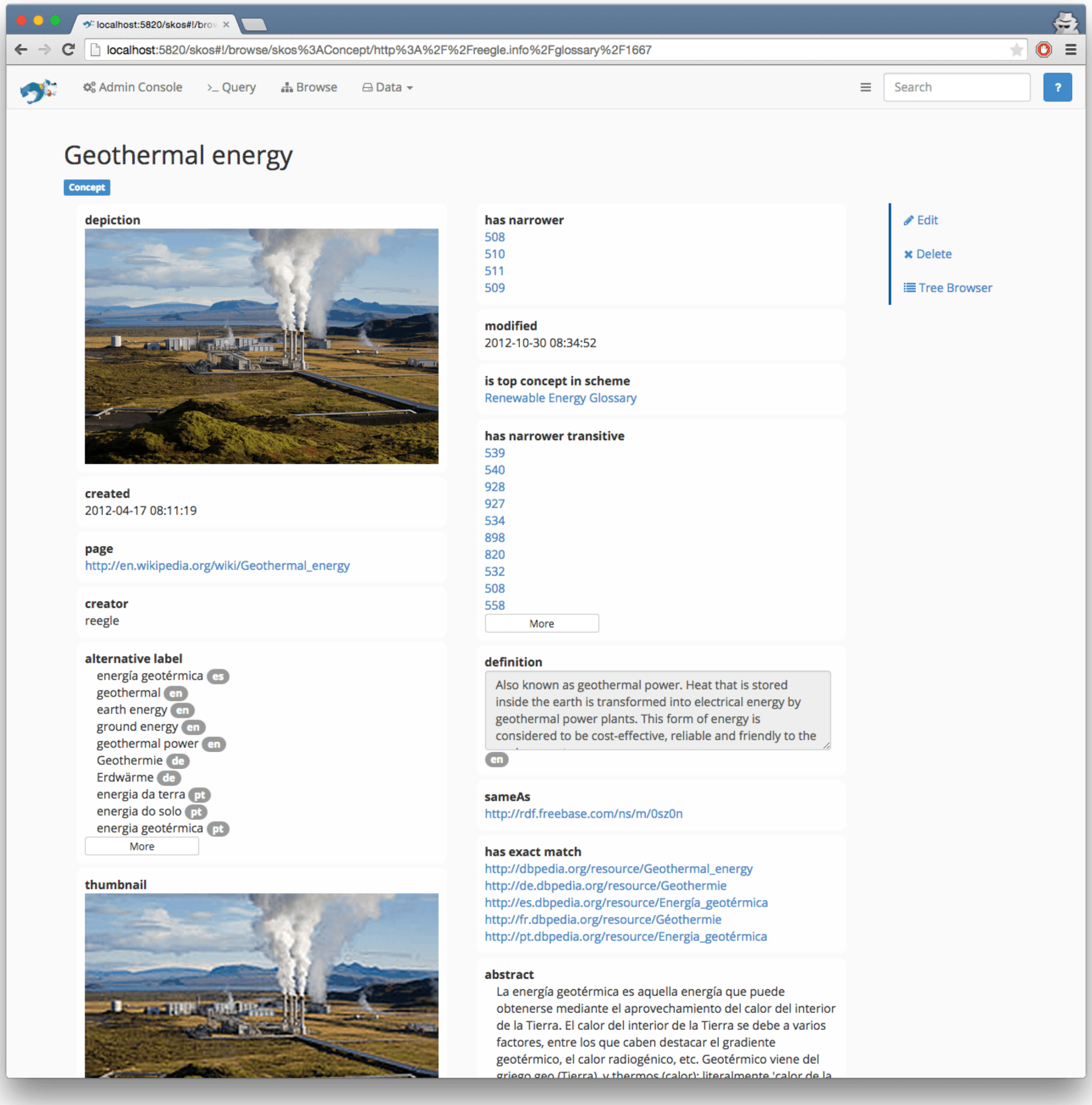
Databases

Name	Status
bsbm1m	online
collections	online
collections2	online
ng	online
nodeDB	online
nodeDBReasoning	online
skos	online
sldc	online
testdb	online
tinkerpop	online

Users

User name	Roles
admin	*No roles assigned*
anonymous	reader

Browsing the Graph



localhost:5820/skos#!/browse/skos%3AConcept/http%3A%2F%2Freegle.info%2Fglossary%2F1667

Geothermal energy

Concept

depiction

created
2012-04-17 08:11:19

page
http://en.wikipedia.org/wiki/Geothermal_energy

creator
reegle

alternative label
energía geotérmica es
geothermal en
earth energy en
ground energy en
geothermal power en
Geothermie de
Erdwärme de
energia da terra pt
energia do solo pt
energia geotérmica pt

thumbnail

has narrower
508
510
511
509

modified
2012-10-30 08:34:52

is top concept in scheme
Renewable Energy Glossary

has narrower transitive
539
540
928
927
534
898
820
532
508
558

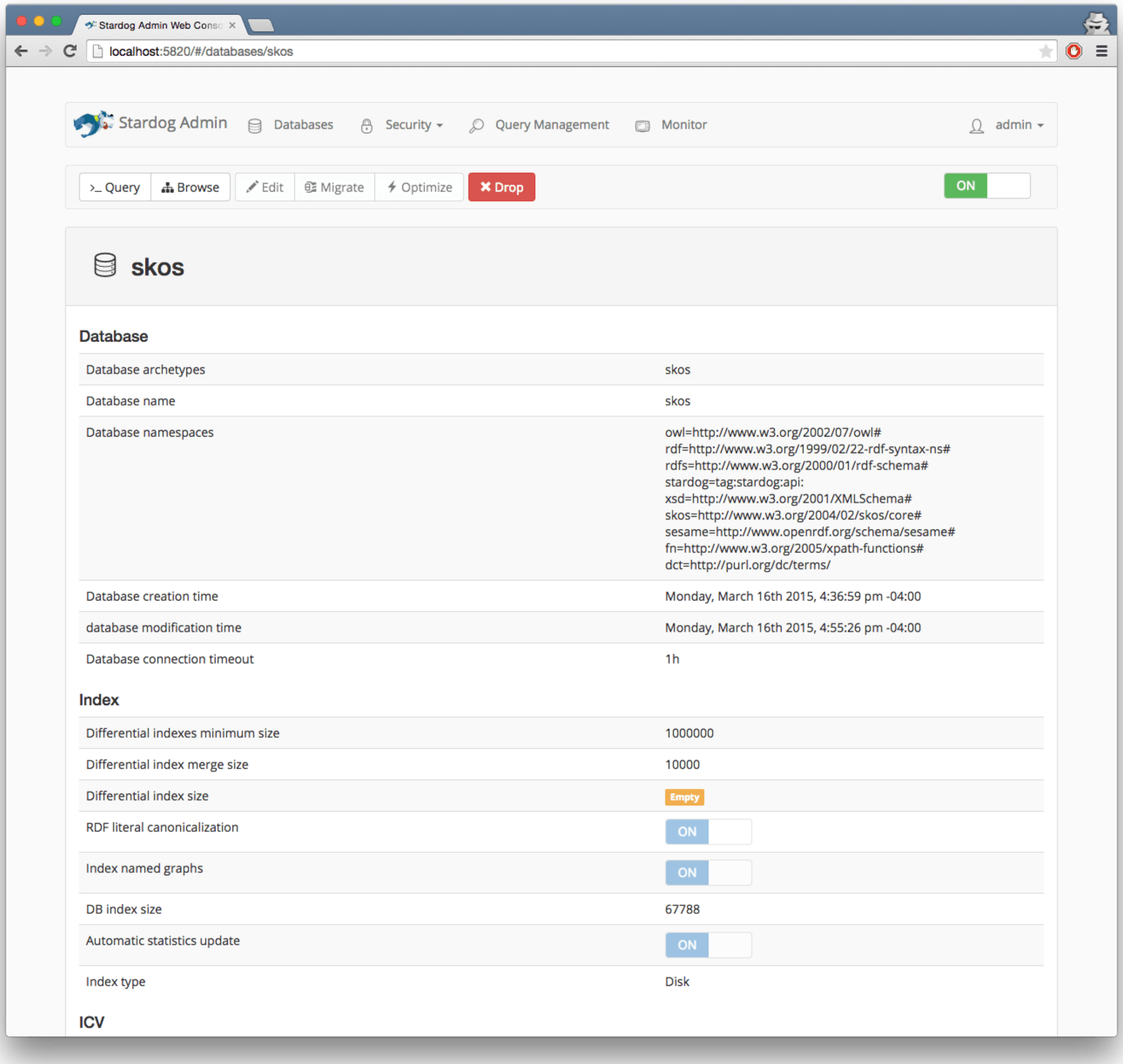
definition
Also known as geothermal power. Heat that is stored inside the earth is transformed into electrical energy by geothermal power plants. This form of energy is considered to be cost-effective, reliable and friendly to the
en

sameAs
http://rdf.freebase.com/ns/m/0sz0n

has exact match
http://dbpedia.org/resource/Geothermal_energy
http://de.dbpedia.org/resource/Geothermie
http://es.dbpedia.org/resource/Energía_geotérmica
http://fr.dbpedia.org/resource/Géothermie
http://pt.dbpedia.org/resource/Energia_geotérmica

abstract
La energía geotérmica es aquella energía que puede obtenerse mediante el aprovechamiento del calor del interior de la Tierra. El calor del interior de la Tierra se debe a varios factores, entre los que caben destacar el gradiente geotérmico, el calor radiogénico, etc. Geotérmico viene del griego geo (Tierra), y thermos (calor); literalmente 'calor de la

Managing a Database



skos

Database

Database archetypes	skos
Database name	skos

Database namespaces	owl=http://www.w3.org/2002/07/owl#
	rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
	rdfs=http://www.w3.org/2000/01/rdf-schema#
	stardog=tag:stardog:api:
	xsd=http://www.w3.org/2001/XMLSchema#
	skos=http://www.w3.org/2004/02/skos/core#
	sesame=http://www.openrdf.org/schema/sesame#
	fn=http://www.w3.org/2005/xpath-functions#
	dct=http://purl.org/dc/terms/

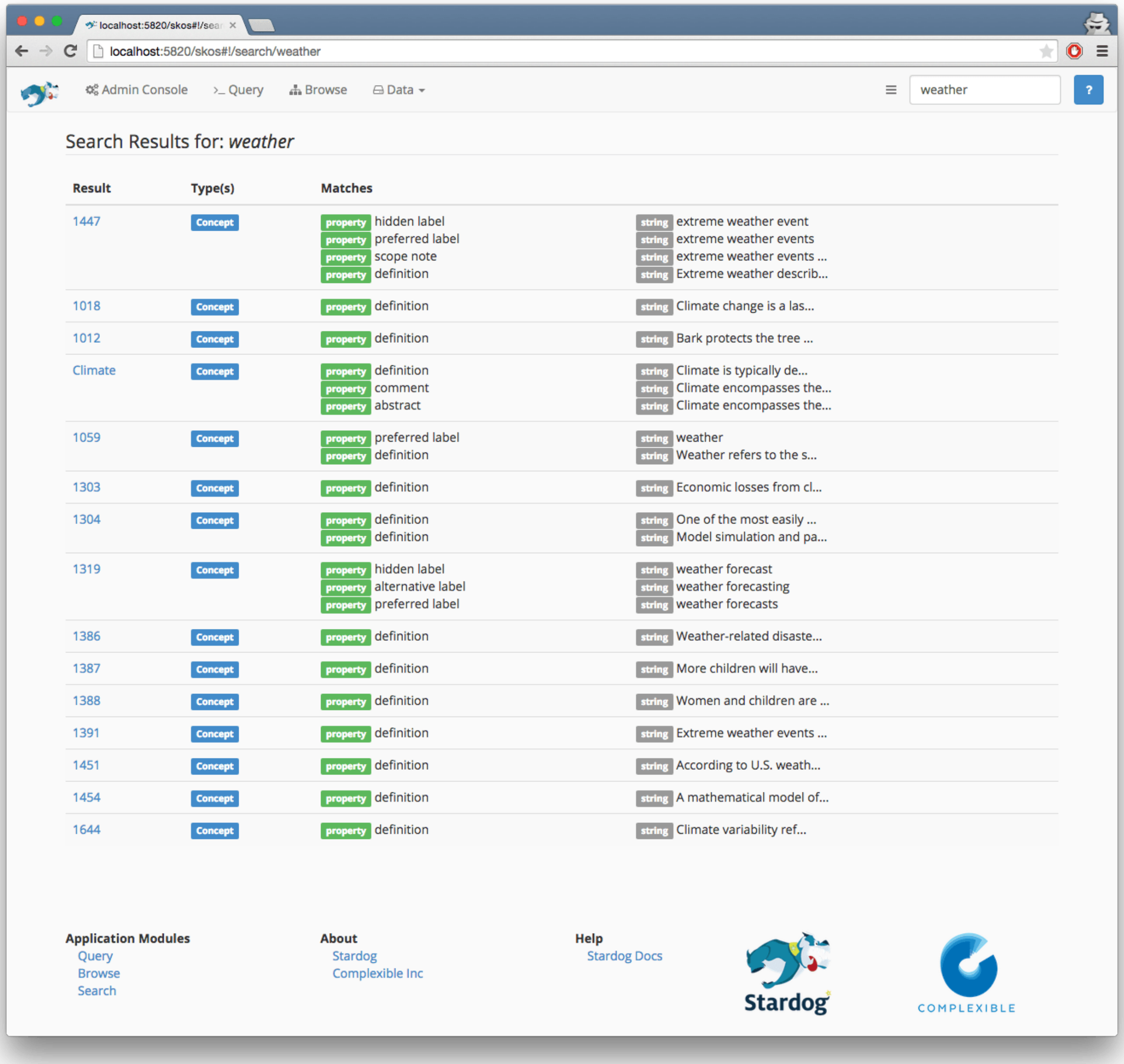
Database creation time	Monday, March 16th 2015, 4:36:59 pm -04:00
database modification time	Monday, March 16th 2015, 4:55:26 pm -04:00
Database connection timeout	1h

Index

Differential indexes minimum size	1000000
Differential index merge size	10000
Differential index size	Empty
RDF literal canonicalization	ON
Index named graphs	ON
DB index size	67788
Automatic statistics update	ON
Index type	Disk

ICV

Searching the Graph


localhost:5820/skos#!/sear
localhost:5820/skos#!/search/weather
Admin Console
>_ Query
Browse
Data
weather
?

Search Results for: *weather*

Result	Type(s)	Matches	
1447	Concept	property hidden label property preferred label property scope note property definition	string extreme weather event string extreme weather events string extreme weather events ... string Extreme weather describ...
1018	Concept	property definition	string Climate change is a las...
1012	Concept	property definition	string Bark protects the tree ...
Climate	Concept	property definition property comment property abstract	string Climate is typically de... string Climate encompasses the... string Climate encompasses the...
1059	Concept	property preferred label property definition	string weather string Weather refers to the s...
1303	Concept	property definition	string Economic losses from cl...
1304	Concept	property definition property definition	string One of the most easily ... string Model simulation and pa...
1319	Concept	property hidden label property alternative label property preferred label	string weather forecast string weather forecasting string weather forecasts
1386	Concept	property definition	string Weather-related disaste...
1387	Concept	property definition	string More children will have...
1388	Concept	property definition	string Women and children are ...
1391	Concept	property definition	string Extreme weather events ...
1451	Concept	property definition	string According to U.S. weath...
1454	Concept	property definition	string A mathematical model of...
1644	Concept	property definition	string Climate variability ref...

Man Pages

Stardog command-line interface is comprehensively documented in `man` pages that ship with Stardog. Those `man` pages are reproduced here in HTML as a convenience to the reader.

Stardog CLI

1. `data add` , `data export` , `data obfuscate` , `data remove` , `data size`

2. `icv convert`, `icv explain`, `icv export`, `icv fix`, `icv validate`
3. `namespace add`, `namespace import`, `namespace list`, `namespace remove`
4. `query execute`, `query explain`, `query obfuscate`, `query search`
5. `reasoning consistency`, `reasoning explain`, `reasoning schema`, `reasoning undo`
6. `vcs commit`, `vcs diff`, `vcs list`, `vcs query`, `vcs revert`, `vcs tag`

Stardog Admin CLI

1. `cluster generate`, `cluster info`, `cluster proxystart`, `cluster proxystop`, `cluster zkstart`, `cluster zkstop`
2. `db backup`, `db copy`, `db create`, `db drop`, `db list`, `db migrate`, `db offline`, `db online`, `db optimize`, `db repair`, `db restore`, `db status`
3. `icv add`, `icv drop`, `icv remove`
4. `license info`
5. `metadata get`, `metadata set`
6. `query kill`, `query list`, `query status`
7. `role add`, `role grant`, `role list`, `role permission`, `role remove`, `role revoke`
8. `server start`, `server status`, `server stop`
9. `user add`, `user addrole`, `user disable`, `user enable`, `user grant`, `user list`, `user passwd`, `user permission`, `user remove`, `user removerole`, `user revoke`

Installing Man Pages Locally

To install the man pages locally in your Unix-like environment:

```
$ cp docs/man/man1/* /usr/local/share/man1
$ cp docs/man/man8/* /usr/local/share/man8
$ mandb
$ man stardog-admin-server-start
```

Administering Stardog

In this chapter we describe the administration of Stardog Server and Stardog databases, including the various command-line programs, configuration options, etc.

Security is an important part of Stardog administration; it's discussed separately (Security).

Command Line Interface

Stardog's command-line interface (CLI) comes in two parts:

1. `stardog-admin` : admininstrative client
2. `stardog` : a user's client

The admin and user's tools operate on local or remote databases, using either HTTP or SNARL protocols. Both of these CLI tools are Unix-only, are self-documenting, and the help output of these tools is their canonical documentation.[7]

Help

To use the Stardog CLI tools, you can start by asking them to display help:

```
stardog help
```

Or:

```
$ stardog-admin help
```

These work too:

```
$ stardog
$ stardog-admin
```

Security Considerations

We divide administrative functionality into two CLI programs for reasons of security: `stardog-admin` will need, in production environments, to have considerably tighter access restrictions than `stardog`.

> **CAUTION**
>
> For usability, Stardog provides a default user "admin" and password "admin" in `stardog-admin` commands if no user or password are given. This is obviously **insecure**; before any serious use of Stardog is contemplated, read the Security section at least twice, and then—minimally—change the administrative password to something we haven't published on the interwebs!

Command Groups

The CLI tools use "command groups" to make CLI subcommands easier to find. To print help for a particular command group, just ask for help:

```
$ stardog help [command_group_name]
```

The command groups and their subcommands:

- **data:** add, remove, export;
- **query:** search, execute, explain, status;
- **reasoning:** explain, consistency;
- **namespace:** add, list, remove;
- **server:** start, stop;
- **metadata:** get, set;
- **user:** add, drop, edit, grant, list, permission, revoke, passwd;
- **role:** add, drop, grant, list, permission, revoke;
- **db:** backup, copy, create, drop, migrate, optimize, list, online, offline, repair, restore, status.

NOTE

See the man pages for the canonical list of commands.

The main help command for either CLI tool will print a listing of the command groups:

```
usage: stardog <command> [ <args> ]

The most commonly used stardog commands are:
    data        Commands which can modify or dump the contents of a database
    help        Display help information
    icv         Commands for working with Stardog Integrity Constraint support
    namespace   Commands which work with the namespaces defined for a database
    query       Commands which query a Stardog database
    reasoning   Commands which use the reasoning capabilities of a Stardog database
    version     Prints information about this version of Stardog

See 'stardog help' for more information on a specific command.
```

To get more information about a particular command, simply issue the help command for it including its command group:

```
$ stardog help query execute
```

Finally, everything here about command groups, commands, and online help works for `stardog-admin`, too:

```
$ stardog reasoning consistency -u myUsername -p myPassword -r myDB

$ stardog-admin db migrate -u myUsername -p myPassword myDb
```

Autocomplete

Stardog also supports CLI autocomplete via `bash` autocompletion. To install autocomplete for bash shell, you'll first want to make sure bash completion is installed:

Homebrew

To install:

```
$ brew install bash-completion
```

To enable, edit `` `.bash\_profile ``:

```
if [ -f `brew --prefix`/etc/bash_completion ]; then
  . `brew --prefix`/etc/bash_completion
fi
```

MacPorts

First, you really should be using Homebrew…ya heard?

If not, then:

```
$ sudo port install bash-completion
```

Then, edit `.bash\_profile`:

```
if [ -f /opt/local/etc/bash_completion ]; then
    . /opt/local/etc/bash_completion
fi
```

Ubuntu

And for our Linux friends:

```
$ sudo apt-get install bash-completion
```

Fedora

```
$ sudo yum install bash-completion
```

All Platforms

Now put the Stardog autocomplete script— `stardog-completion.sh` —into your `bash\_completion.d` directory, typically one of `/etc/bash_completion.d, /usr/local/etc/bash_completion.d or ~/bash_completion.d.`

Alternately you can put it anywhere you want, but tell `.bash_profile` about it:

```
source ~/.stardog-completion.sh
```

How to Make a Connection String

You need to know how to make a connection string to talk to a Stardog database. A connection string may consist solely of the **database name** in cases where

1. Stardog is listening on the standard port(s);
2. SNARL is enabled; and
3. the command is invoked on the same machine where the server is running.

In other cases, a "fully qualified" connection string, as described below, is required.

Further, the connection string is now assumed to be the first argument of any command that requires a connection string. Some CLI subcommands require a Stardog connection string as an argument to identify the server and database upon which operations are to be performed.

Connection strings are URLs and may either be local to the machine where the CLI is run or they may be on some other remote machine.

There are two URL schemes recognized by Stardog:

1. `http://`
2. `snarl://`

The former uses Stardog’s (extended) version of SPARQL Protocol; the latter uses Stardog’s native data access protocol, called SNARL.

NOTE

> `stardog-admin` and `stardog` works with HTTP or SNARL Protocol, interchangeably. SNARL is faster than HTTP in cases where payloads to and from the server are relatively **small**; for payloads that are large, raw transfer time dominates and there isn't much or any difference in performance between them.

Example Connection Strings

To make a connection string, you need to know the URL scheme; the machine name and port Stardog Server is running on; any (optional) URL path to the database;[8] and the name of the database:

```
{scheme}{machineName}:{port}/{databaseName};{connectionOptions}
```

Here are some example connection strings:

```
snarl://server/billion-triples-punk
http://localhost:5000/myDatabase
http://169.175.100.5:1111/myOtherDatabase;reasoning=true
snarl://stardog:8888/the_database
snarl://localhost:1024/db1;reasoning=true
```

Using the default ports for SNARL and HTTP protocols simplifies connection strings. `connectionOptions` are a series of `;` delimited key-value pairs which themselves are `=` delimited. Key names must be **lowercase** and their values are case-sensitive. Finally, in the case where the scheme is SNARL, the machine is "localhost", and the port is the default SNARL port, a connection string may consist of the "databaseName" only.

Server Admin

Stardog Server is multi-protocol, supporting SNARL and HTTP. The default port for SNARL is **5820**; the default port for HTTP is **5822. All administrative functions work over SNARL or HTTP protocols.**

Upgrading Stardog Server

The process of installation is pretty simple; see the Quick Start Guide for details.

But how do we easily upgrade between versions? The key is judicious use of `STARDOG_HOME`. Best practice is to keep installation directories for different versions separate and use a `STARDOG_HOME` in another location for storing databases.[9] Once you set your `STARDOG_HOME` environment variable to point to this directory, you can simply stop the old version and start the new version without copying or moving any files. You can also specify the home directory using the `--home` argument when starting the server.

HTTP & SNARL Server Unification

To use any of these commands against a remote server, pass a global `--server` argument with an HTTP or SNARL URL.

> **NOTE** If you are running `stardog-admin` on the same machine where Stardog Server is running, and you're using the default protocol ports, then you can omit the `--server` argument and simply pass a database name via `-n` option. Most of the following commands assume this case for the sake of exposition.

Server Security

See the Security section for information about Stardog's security system, secure deployment patterns, and more.

Configuring Stardog Server

> **NOTE** The properties described in this section control the behavior of the Stardog Server (whether HTTP or SNARL protocols are in use); to set properties or other metadata on individual Stardog **databases**, see Database Admin.

Stardog Server's behavior can be configured via the JVM arg `stardog.home`, which sets Stardog Home, overriding the value of `STARDOG_HOME` set as an environment variable. Stardog Server's behavior can also be configured via a `stardog.properties`—which is a Java Properties file—file in `STARDOG_HOME`. To change the behavior of a running Stardog Server, it is necessary to restart it.

Configuring Temp Space

Stardog uses the value of the JVM argument `java.io.tmpdir` to write temporary files for many different operations. If you want to configure temp space to use a particular disk volume or partition, use the `java.io.tmpdir` JVM argument on Stardog startup.

Bad (or at least very weird) things are guaranteed to happen if this part of the filesystem runs out of (or even very low on) free disk space. Stardog will delete temporary files when they're no longer needed. But Stardog admins should configure their monitoring systems to make sure that free disk space is always available, both on `java.io.tmpdir` and on the disk volume that hosts `STARDOG_HOME`.[10]

Stardog Configuration

The following twiddly knobs for Stardog Server are available in `stardog.properties`:[11]

1. `query.all.graphs`: Controls what data Stardog Server evaluates queries against; if `true`, it will query over the default graph and the union of all named graphs; if `false` (the default), it will query only over the default graph.
2. `query.timeout`: Sets the upper bound for query execution time that's inherited by all databases unless explicitly overriden. See Managing Queries section below for details.
3. `logging.[access,audit].[enabled,type,file]`: Controls whether and how Stardog logs server events; described in detail below.
4. `logging.slow_query.enabled`, `logging.slow_query.time`, `logging.slow_query.type`: The three slow query logging options are used in the following way. To enable logging of slow queries, set `enabled` to `true`. To define what counts as a "slow" query, set `time` to a time duration value (positive integer plus "h", "m", "s", or "ms" for hours, minutes, seconds, or milliseconds respectively). To determine the type of logging, set `type` to `text` (the default) or `binary`. **To state the obvious explicitly, a `logging.slow_query.time` that exceeds the value of `query.timeout` will result in null logs.**

5. `database.connection.timeout.ms` : Controls how long, in milliseconds, connections may idle before being automatically closed by the server.
6. `http.max.request.parameters` : Default is 1024; any value smaller than `Integer.MAX_VALUE` may be provided. Useful if you have lots of named graphs.
7. `database.connection.timeout` : The amount of time a connection to the database can be open, but inactive, before being automatically closed to reclaim the resources. The timeout values specified in the property file should be a positive integer followed by either letter `h` (for hours), letter `m` (for minutes), letter `s` (for seconds), or letters `ms` (for milliseconds). Example intervals: `1h` for 1 hour, `5m` for 5 minutes, `90s` for 90 seconds, `500ms` for 500 milliseconds. Default value is `1h` .
8. `password.length.min` : Sets the password policy for the minimum length of user passwords, the value can't be lower than `password.length.min` or greater than `password.length.max` . Default: `4` .
9. `password.length.max` : Sets the password policy for the maximum length of user passwords. Default: `1024` .
10. `password.regex` : Sets the password policy of accepted chars in user passwords, via a Java regular expression. Default: `[\\w@#$%]+`
11. `security.named.graphs` : Sets named graph security on globally. Default: `false` .

Starting & Stopping the Server

> **NOTE** Unlike the other `stardog-admin` subcommands, starting the server may only be run locally, i.e., on the same machine the Stardog Server is will run on.

The simplest way to start the server—running on the default port, detaching to run as a daemon, and writing `stardog.log` to the current working directory— is

```
$ stardog-admin server start
```

To specify parameters:

```
$ stardog-admin server start --logfile mystardog.log --port=8080
```

The port can be specified using the property `--port`. The HTTP interface can be disabled by using the flag `--no-http` and the SNARL interface via `--no-snarl`.

To shut the server down:

```
$ stardog-admin server stop
```

If you started Stardog on a port other than the default, or want to shut down a remote server, you can simply use the `--server` option to specify the location of the server to shutdown.

By default Stardog will bind it's server to `0.0.0.0`. You can specify a different network interface for Stardog to bind to using the `--bind` property of `server start`.

Server Monitoring with Watchdog & JMX

Stardog's JMX implementation is called Watchdog. In addition to providing some basic JVM information, Watchdog also exports information about the Stardog DBMS configuration as well as stats for all of the databases within the system, such as the total number of open connections, size, and average query time.

Accessing Watchdog

To access Watchdog, you can simply use a tool like VisualVM or JConsole to attach to the process running the JVM, or connect directly to the JMX server. You can also access information from Watchdog in the web console for the database, or by performing a `GET` on `/watchdog` which will return a simple JSON object containing the information available via JMX.

Configuring Watchdog

By default, Watchdog will bind an RMI server for remote access on port `5833`. If you want to change which port Watchdog binds the remote server to, you can set the property `watchdog.port` via `stardog.properties`. If you wish to disable remote access to JMX, you can set `watchdog.remote.access` to `false` in `stardog.properties`. Finally, if you wish to disable Watchdog completely, set `watchdog.enabled` to `false` in `stardog.properties`.

Locking Stardog Home

Stardog Server will lock `STARDOG_HOME` when it starts to prevent synchronization errors and other nasties if you start more than one Stardog Server with the same `STARDOG_HOME`. If you need to run more than one Stardog Server instance, choose a different `STARDOG_HOME` or pass a different value to `--home`.

Access & Audit Logging

See the `stardog.properties` file (in the distribution) for a complete discussion of how access and audit logging work in Stardog Server. Basically, audit logging is a superset of the events in access logging. Access logging covers the most often required logging events; you should consider enabling audit logging if you really need to log **every** server event. Logging generally doesn’t have much impact on performance; but the safest way to insure that impact is negligible is to log to a separate disk (or to a centralized logging server, etc.).

The important configuration choices are whether logs should be binary or plain text (both based on ProtocolBuffer message formats); the type of logging (audit or access); the logging location (which may be "off disk" or even "off machine") Logging to a centralized logging facility requires a Java plugin that implements the Stardog Server logging interface; see Java Programming for more information; and the log rotation policy (file size or time).

Slow query logging is also available. See the Managing Queries section below.

Database Admin

Stardog is a multi-tenancy system and will happily provide access to multiple, distinct databases.

Configuring a Database

To administer a Stardog database, some config options must be set at creation time; others may be changed subsequently and some may never be changed. All of the config options have sensible defaults (except, obviously, for the database name), so you don’t have to twiddle any of the knobs till you really need to.

To configure a database, use the `metadata-get` and `metadata-set` CLI commands. See Man Pages for the details.

Configuration Options

1. Table of Configuration Options

Option	Mutable	Default	API
`preserve.bnode.ids`	Yes	`true`	DatabaseOptions.PRESERVE_BNODE_
Determines how the Stardog parser handles bnode identifiers that may be present in RDF input. If this prope enabled (i.e., `TRUE`), parsing and data loading performance are improved; but the other effect is that if distir files use (randomly or intentionally) the same bnode identifier, that bnode will point to one and the same no database. If you have input files that use explicit bnode identifiers, and multiple files may use the same bnoc idenitifers, and you don't want those bnodes to be smushed into a single node in the database, then this cor option should be disabled (set to `FALSE`).			
`database.archetypes`	Yes		DatabaseOptions.ARCHETYPES
The name of one or more database archetypes.			
`database.name`	No		DatabaseOptions.NAME
A database name, the legal value of which is is given by the regular expression `[A-Za-z]{1}[A-Za-z0-9_-]`			
`database.namespaces`	Yes	`rdf, rdfs, xsd, owl, stardog`	DatabaseOptions.NAMESPACES
Sets the default namespaces for new databases.			
`database.online`	No	`true`	DatabaseOptions.ONLINE
The status of the database: online or offline. It may be set so that the database is created initially in online or status; subsequently, it can't be set directly but only by using the relevant admin commands.			

Option	Mutable	Default	API
`icv.active.graphs`	No	`default`	ICVOptions.ICV_ACTIVE_GRAPHS
Specifies which part of the database, in terms of named graphs, is checked with IC validation. Set to `tag:stardog:api:context:all` to validate all the named graphs in the database; otherwise, the legal valu `icv.active.graphs` is a comma-separated list of named graph identifiers.			
`icv.consistency.automatic`	No	`false`	ICVOptions.ICV_CONSISTENCY_AUTOM
Enables automatic ICV consistency check as part of transactions.			
`icv.enabled`	Yes	`false`	ICVOptions.ICV_ENABLED
Determines whether ICV is active for the database; if true, all database mutations are subject to IC validation "guard mode").			
`icv.reasoning.enabled`	Yes	`false`	ICVOptions.ICV_REASONING_ENABLED
Determines if reasoning is used during IC validation.			
`index.connection.timeout`	Yes	`3,600,000`	IndexOptions.INDEX_CONNECTION_TI
Determines timeout value for open connections.			
`index.differential.enable.limit`	Yes	`1,000,000`	IndexOptions.DIFF_INDEX_MIN_LIMIT
Sets the minimum size of the Stardog database before differential indexes are used. The legal value is an inte			
`index.differential.merge.limit`	Yes	`10,000`	IndexOptions.DIFF_INDEX_MAX_LIMIT
Sets the size in number of RDF triples before the differential indexes are merged to the main indexes. The leg an integer.			
`index.literals.canonical`	No	`true`	IndexOptions.CANONICAL_LITERALS
Enables RDF literal canonicalization.			

Option	Mutable	Default	API
`index.named.graphs`	No	`true`	IndexOptions.INDEX_NAMED_GRAPHS
Enables optimized index support for named graphs; speeds SPARQL query evaluation with named graphs at some overhead for database loading and index maintenance.			
`index.persist`	Yes	`false`	IndexOptions.PERSIST
Enables persistent indexes.			
`index.persist.sync`	Yes	`true`	IndexOptions.SYNC
Determines whether memory indexes are synchronously or asynchronously persisted to disk with respect to transaction.			
`index.statistics.update.automatic`	Yes	`true`	IndexOptions.AUTO_STATS_UPDATE
Determines whether statistics are maintained automatically.			
`index.type`	No	`disk`	IndexOptions.INDEX_TYPE
The legal value of `index.type` is the string "disk" or "memory" (case-insensitive).			
`query.timeout`	Yes		DatabaseOptions.QUERY_TIMEOUT
Determines max execution time for query evaluation.			
`reasoning.consistency.automatic`	Yes	`false`	ReasoningOptions.CONSISTENCY_AUT
Enables automatic consistency checking with respect to a transaction.			
`reasoning.punning.enabled`	No	`false`	ReasoningOptions.PUNNING_ENABLE
Enables punning.			
`reasoning.schema.graphs`	Yes	`*`	ReasoningOptions.SCHEMA_GRAPHS

Option	Mutable	Default	API
Determines which, if any, named graph or graphs contains the "tbox", i.e., the schema part of the data. The le a comma-separated list of named graph identifiers, including (optionally) the special names, `tag:stardog:api:context:default` and `tag:stardog:api:context:all` , which represent the default the union of all named graphs and the default graph, respectively. In the context of database configurations Stardog will recognize `default` and `*` as short forms of those URIs, respectively.			
`reasoning.type`	Yes	`SL`	
Specifies the reasoning type associated with the database; legal values are `SL` , `RL` , `QL` , `EL` , `DL` , `RDFS` , an			
`reasoning.approximate`	Yes	`false`	ReasoningOptions.APPROXIMATE
Enables approximate reasoning. With this flag enabled Stardog will approximate an axiom that is outside the Stardog supports and normally ignored. For example, an equivalent class axiom might be split into two subc axioms and only one subclass axiom is used.			
`search.enabled`	Yes	`false`	SearchOptions.SEARCHABLE
Enables semantic search for the database.			
`strict.parsing`	No	`true`	DatabaseOptions.STRICT_PARSING
Controls whether Stardog parses RDF strictly (`true` , the default) or loosely (`false`)			
`search.reindex.mode`	Yes	`async`	
Determins how search indexes are maintained. The legal value of `search.reindex.mode` is one of the strin `async` (case insensitive) or a legal Quartz cron expression.			
`transactions.durable`	Yes	`false`	DatabaseOptions.TRANSACTIONS_DU
Enables durable transactions.			
`query.all.graphs`	Yes	`false`	DatabaseOptions.QUERY_ALL_GRAPH

Option	Mutable	Default	API
Determines what data the database evaluates queries against; if `true` , it will query over the default graph a union of all named graphs; if `false` (the default), it will query only over the default graph. This database op overrides any global server settings.			

A Note About Database Status

A database must be set to `offline` status before most configuration parameters may be changed. Hence, the normal routine is to set the database offline, change the parameters, and then set the database to online. All of these operations may be done programmatically from CLI tools, such that they can be scripted in advance to minimize downtime. In a future version, we will allow some properties to be set while the database remains online.

Managing Database Status

Databases are either online or offline; this allows database maintenance to be decoupled from server maintenance.

Online and Offline

Databases are put online or offline synchronously: these operations block until other database activity is completed or terminated. See `stardog-admin help db` for details.

Examples

To set a database from offline to online:

```
$ stardog-admin db offline myDatabase
```

To set the database online:

```
$ stardog-admin db online myDatabase
```

If Stardog Server is shutdown while a database is offline, the database will be offline when the server restarts.

Creating a Database

Stardog databases may be created locally or remotely; but, of course, performance is better if data files don't have to be transferred over a network during creation and initial loading. See the section below about loading compressed data. All data files, indexes, and server metadata for the new database will be stored in Stardog Home. Stardog won't create a database with the same name as an existing database. Stardog database names must conform to the regular expression, `[A-Za-z]{1}[A-Za-z0-9_-]`.

> **NOTE** There are four reserved words that may not be used for the names of Stardog databases: `system`, `admin`, `watchdog`, and `docs`.

Minimally, the only thing you must know to create a Stardog database is a database **name**; alternately, you may customize some other database parameters and options depending on anticipated workloads, data modeling, and other factors.

See `stardog-admin help db create` for all the details including examples.

Database Archetypes

Stardog database archetypes are a new feature in 2.0. A database archetype is a named, vendor-defined or user-defined bundle of data and functionality to be applied at database-creation time. Archetypes are primarily for supporting various data standards or toolchain configurations in a simple way.

For example, the SKOS standard from W3C defines an OWL vocabulary for building taxonomies, thesauruses, etc. SKOS is made up by a vocabulary, some constraints, some kinds of reasoning, and (typically) some SPARQL queries. If you are developing an app that uses SKOS, without Stardog's SKOS archetype, you are responsible for assembling all that SKOS stuff yourself. Which is tedious, error-prone, and not very rewarding—even when it's done right the first time.

Rather than putting that burden on Stardog users, we've created database archetypes as a mechanism to collect these "bundles of stuff" which, as a developer, you can then simply attach to a particular database.

The last point to make is that archetypes are composable: you can mix-and-match them at database creation time as needed.

Stardog supports two database archetypes out-of-the-box: PROV and SKOS.

SKOS Archetype

The SKOS archetype is for databases that will contain SKOS data, and includes the SKOS schema, SKOS constraints using Stardog's Integrity Constraint Validation, and some namespace-prefix bindings.

PROV Archetype

The PROV archetype is for databases that will contain PROV data, and includes the SKOS schema, SKOS constraints using Stardog's Integrity Constraint Validation, and some namespace-prefix bindings.

Archetypes are composable, so you can use more of them and they are intended to be used alongside **your domain data**, which may include as many other schemas, ontologies, etc. as are required.

User-defined Archetypes

Please see the Stardog Examples repository on Github for an example that shows how to create your own Stardog archetype.

Database Creation Templates

As a boon to the overworked admin or devops peeps, Stardog Server supports database creation templates: you can pass a Java Properties file with config values set and with the values (typically just the database name) that are unique to a specific database passed in CLI parameters.

Examples

To create a new database with the default options by simply providing a name and a set of initial datasets to load:

```
$ stardog-admin db create -n myDb input.ttl another_file.rdf moredata.rdf.gz
```

Datasets can be loaded later as well. To create (in this case, an empty) database from a template file:

```
$ stardog-admin db create -c database.properties
```

At a minimum, the configuration file must have a value for `database.name` option.

If you only want to change only a few configuration options you can directly provide the values for these options in the CLI args as follows:

```
$ stardog-admin db create -n db -o icv.enabled=true icv.reasoning.enabled=true --
input.ttl
```

Note that "--" is used in this case when "-o" is the last option to delimit the value for "-o" from the files to be bulk loaded.

Please refer to the CLI help for more details of the `db create` command.

Database Create Options

2. Table of Options for Stardog's `create` *command*

Name	Description	Arg values	Default
`--durable`, `-d`	If present, sets all mutation operations to database as transactionally durable; durability increases the cost of all mutation operations.		`false`

Name	Description	Arg values	Default
`--type` , `-t`	Specifies the kind of database indexes: memory or disk	`M` , `D`	disk
`--index-triples-only` , `-i`	Specifies that the database's indexes should be optimized for RDF triples only		`false`

Repairing a Database

If an I/O error or an index exception occurs while querying a DB, the DB might be corrupted and repaired with the repair command. If the errors occur during executing admin commands, then the system DB might have been corrupted. System database corruptions can also cause other problems including authorization errors.

This command needs exclusive access to your Stardog home directory and therefore requires the Stardog Server not to be running. This also means that the command can only be run on the machine where the Stardog home directory is located, and you will not be able to start the Stardog Server while this command is running.

NOTE The repair process can take considerable time for large databases.

If the built-in Stardog system database is corrupted, then you can use the database name `system` as the repair argument. To repair the database myDB:

```
$ stardog-admin db repair myDB
```

To repair the system database:

```
$ stardog-admin db repair system
```

Backing Up and Restoring

Stardog includes both physical and logical backup utilities; logical backups are accomplished using the `export` CLI command. Physical backups and restores are accomplished using `stardog-admin db backup` and `stardog-admin db restore` commands, respectively.

These tools perform physical backups, including database metadata, rather than logical backups via some RDF serialization. They are **native** Stardog backups and can only be restored with Stardog tools. Backup may be accomplished while a database is online; backup is performed in a read transaction: reads and writes may continue, but writes performed during the backup are not reflected in the backup.

See the man pages for `backup` and `restore` for details.

Backup

`stardog-admin db backup` assumes a default location for its output, namely, `$STARDOG_HOME/.backup`; that default may be overriden by passing a `-t` or `--to` argument. Backup sets are stored in the backup directory by database name and then in date-versioned subdirectories for each backup volume. Of course you can use a variety of OS-specific options to accomplish remote backups over some network or data protocol; those options are left as an exercise for the admin.

To backup a Stardog database called `foobar`:

```
$ stardog-admin db backup foobar
```

To perform a remote backup, for example, pass in a specific directory that may be mounted in the current OS namespace via some network protocol, thus:

```
$ stardog-admin db backup --to /my/network/share/stardog-backups foobar
```

NOTE Stardog’s backup/restore approach is optimized for minimizing the amount of time it takes to backup a database; the tradeoff is with restore performance.

Restore

To restore a Stardog database from a Stardog backup volume, simply pass a fully-qualfied path to the volume in question. The location of the backup should be the full path to the backup, not the location of the backup directory as specified in your Stardog configuration. There is no need to specify the name of the database to restore.

To restore a database from its backup:

```
$ stardog-admin db restore $STARDOG_HOME/.backups/myDb/2012-06-21
```

One-time Database Migrations for Backup

The backup system cannot directly backup atbases created in versions before 2.1. These databases must be explicitly migrated in order to use the new backup system; this is a one-time operation per atbase and is accomplished by running

ource,bash]

```
$ stardog-admin db migrate foobar
```

in order to migrate a database called `foobar`. Again, this is a one-time operation only and all databases created with 2.1 (or later) do not require it.

Namespace Prefix Bindings

SPARQL queries can be verbose; but at least the `PREFIX` declarations in the prologue of each query are easy to screw up! Stardog allows database administrators to persist and manage custom namespace prefix bindings:

1. At database creation time, if data is loaded to the database that contains namespace prefixes, then those are persisted for the life of the database. Any subsequent queries to the database may simply omit the `PREFIX` declarations:

   ```
   $ stardog query myDB "select * {?s rdf:type owl:Class}"
   ```

2. To add new bindings, use the `namespace` subcommand in the CLI:

   ```
   $ stardog namespace add myDb --prefix ex --uri 'http://example.org/test#'
   ```

3. To modify an existing binding, delete the existing one and then add a new one:

   ```
   $ stardog namespace remove myDb --prefix ex
   ```

4. Finally, to see all of the existing namespace prefix bindings:

   ```
   $ stardog namespace list myDB
   ```

If no files are used during database creation, or if the files do not define any prefixes (e.g. NTriples), then the "Big Four" default prefixes are stored: `RDF, RDFS, XSD`, and `OWL`.

When executing queries in the CLI, the default table format for SPARQL `SELECT` results will use the bindings as qnames. SPARQL `CONSTRUCT` query output (including export) will also use the stored prefixes. To reiterate, namespace prefix bindings are **per database**, not global.

Index Strategies

By default Stardog builds extra indexes for named graphs. These additional indexes are used when SPARQL queries specify datasets using `FROM` and `FROM NAMED`. With these additional indexes, better statistics about named graphs are also computed.

Stardog may also be configured to create and to use fewer indexes, if the database is only going to be used to store RDF triples—that is to say, if the database will not be used to store named graph information. In this mode, Stardog will maintain fewer indexes, which will result in faster database creation and faster updates without compromising query answering performance. In such databases, quads (that is: triples with named graphs or contexts specified) may still be added to these database at any time, but query performance may degrade in such cases.

To create a database which indexes only RDF triples, set the option `index.named.graphs` to `false` at database creation time. The CLI provides a shorthand option, `-i` or `--index-triples-only`, which is equivalent.

NOTE

> This option can only be set at database creation time and cannot be changed later without rebuilding the database; use this option with care.

Differential Indexes

While Stardog is generally biased in favor of read performance, write performance is also important in many applications. In order to increase write performance, Stardog may be used, optionally, with a **differential index**.

Stardog's differential index is used to persist additions and removals separately from the main indexes, such that updates to the database can be performed faster. Query answering takes into consideration all the data stored in the main indexes and the differential index; hence, query answers are computed as if all the data is stored in the main indexes.

There is a slight overhead for query answering with differential indexes if the differential index size gets too large. For this reason, the differential index is merged into the main indexes when its size reaches `DIFF_INDEX_MAX_LIMIT`. There is no benefit of differential indexes if the main index itself is small. For this reason, the differential index is not used until the main index size reaches `DIFF_INDEX_MAX_LIMIT`.

In most cases, the default value of the `DIFF_INDEX_MAX_LIMIT` parameter will work fine and doesn't need to be changed. The corollary is that you shouldn't change this value in a production system till you've tested the effects of a change in a non-production system.

Loading Compressed Data

Stardog supports loading data from compressed files directly: there's no need to uncompress files before loading. Loading compressed data is the recommended way to load large input files. Stardog supports GZIP and ZIP compressions natively.[12]

GZIP and BZIP2

A file passed to `create` will be treated as compressed if the file name ends with `.gz` or `.bz2`. The RDF format of the file is determined by the penultimate extension. For exammple, if a file named `test.ttl.gz` is used as input, Stardog will perform GZIP decompression during loading and parse the file with Turtle parser. All the formats supported by Stardog (RDF/XML, Turtle, Trig, etc.) can be used with compression.

ZIP

The ZIP support works differently since zipped files can contain multiple files. When an input file name ends with `.zip`, Stardog performs ZIP decompression and tries to load all the files inside the ZIP file. The RDF format of the files inside the zip is determined by their file names as usual. If there is an unrecognized file extension (e.g. '.txt'), then that file will be skipped.

Dropping a Database

This command removes a database and all associated files and metadata. This means all files on disk pertaining to the database will be deleted, so only use `drop` when you're certain! Databases must be offline in order to be dropped.

It takes as its only argument a valid database name. For example,

```
$ stardog-admin db drop my_db
```

Using Integrity Constraint Validation

Stardog supports integrity constraint validation as a data quality mechanism via closed world reasoning. Constraints can be specified in OWL, SWRL, and SPARQL. Please see the Validating Constraints section for more about using ICV in Stardog.

The CLI `icv` subcommand can be used to add, delete, or drop all constraints from an existing database. It may also be used to validate an existing database with constraints that are passed into the `icv` subcommand; that is, using different constraints than the ones already associated with the database.

For details of ICV usage, see `stardog help icv` and `stardog-admin help icv`. For ICV in transacted mutations of Stardog databases, see the database creation section above.

Migrating a Database

The `migrate` subcommand migrates an older Stardog database to the latest version of Stardog. Its only argument is the name of the database to migrate. `migrate` won't necessarily work between arbitrary Stardog version, so before upgrading check the release notes for a new version carefully to see whether migration is required or possible.

```
$ stardog-admin db migrate myDatabase
```

will update `myDatabase` to the latest database format.

Getting Database Information

You can get some information about a database by running the following command:

```
$ stardog-admin metadata get my_db_name
```

This will return all the metadata stored about the database, including the values of configuration options used for this database instance. If you want to get the value for a specific option then you can run the following command:

```
$ stardog-admin metadata get -o index.named.graphs my_db_name
```

Managing Queries

Stardog includes the capability to manage running queries according to configurable policies set at run-time; this capability includes support for **listing** running queries; **deleting** running queries; **reading** the status of a running query; **killing** running queries that exceed a time threshold automatically; and **logging** slow queries for analysis.

Stardog is pre-configured with sensible **server-wide** defaults for query management parameters; these defaults may be overridden or disabled per database.

Configuring Query Management

For many uses cases the default configuration will be sufficient. But you may need to tweak the timeout parameter to be longer or shorter, depending on the hardware, data load, queries, throughput, etc. The default configuration has a server-wide query timeout value of `query.timeout`, which is inherited by all the databases in the server. You can customize the server-wide timeout value and then set per-database custom values, too. Any database without a custom value inherits the server-wide value. To disable query timeout, set `query.timeout` to `0`.

Listing Queries

To see all running queries, use the `query list` subcommand:

```
$ stardog-admin query list
```

The results are formatted tabularly:

```
+----+----------+-------+--------------+
| ID | Database | User  | Elapsed time |
+----+----------+-------+--------------+
| 2  | test     | admin | 00:00:20.165 |
| 3  | test     | admin | 00:00:16.223 |
| 4  | test     | admin | 00:00:08.769 |
+----+----------+-------+--------------+

3 queries running
```

You can see which user owns the query (superuser’s can see all running queries), as well as the elapsed time and the database against which the query is running. The ID column is the key to deleting queries.

Deleting Queries

To delete a running query, simply pass its ID to the `query kill` subcommand:

```
$ stardog-admin query kill 3
```

The output confirms the query kill completing successfully:

```
Query 3 killed successfully
```

Automatically Killing Queries

For production use, especially when a Stardog database is exposed to arbitrary query input, some of which may not execute in an acceptable time period, the automatic query killing feature is useful. It will protect a Stardog Server from queries that consume too many resources.

Once the execution time of a query exceeds the value of `query.timeout`, the query will be killed automatically.[13] The client that submitted the query will receive an error message. The value of `query.timeout` may be overriden by setting a different value (smaller or longer) in database options. To disable, set to `query.timeout` to `0`.

The value of `query.timeout` is a positive integer concated with a letter, interpreted as a time duration: 'h' (for hours), 'm' (for minutes), 's' (for seconds), or 'ms' (for milliseconds). For example, '1h' for 1 hour, '5m' for 5 minutes, '90s' for 90 seconds, and '500ms' for 500 milliseconds.

The default value of `query.timeout` is five minutes.

Query Status

To see more detail about query in-flight, use the `query status` subcommand:

```
$ stardog-admin query status 1
```

The resulting output includes query metadata, including the query itself:

```
Username: admin
Database: test
Started : 2013-02-06 09:10:45 AM
Elapsed : 00:01:19.187
Query   :
select ?x ?p ?o1 ?y ?o2
   where {
     ?x ?p ?o1.
     ?y ?p ?o2.
     filter (?o1 > ?o2).
    }
order by ?o1
limit 5
```

Slow Query Logging

Stardog does not log slow queries in the default configuration because there isn’t a single value for what counts as a "slow query", which is entirely relative to queries, access patterns, dataset sizes, etc. While slow query logging has very minimal overhead, what counts as a slow query in some context may be quite acceptable in another. See Configuring Stardog Server above for the details.

Protocols and Java API

For HTTP protocol support, see Stardog’s Apiary docs.

For Java, see the Javadocs.

Security and Query Management

The security model for query management is very simple: any user can kill any running query submitted by that user, and a superuser can kill any running query. The same general restriction is applied to query status; you cannot see status for a query that you do not own, and a superuser can see the status of every query.

Managing Search

Stardog’s search service is described in Using Stardog section.

Full-text support for a database is disabled by default but can be enabled at any time by setting the configuration option `search.enabled` to true. For example, you can create a database with full-text support as follows:

```
$ stardog-admin db create -o search.enabled=true -n myDb
```

There are three modes for rebuilding indexes:

1. `sync` : Recompute the search index synchronously with a transacted write.
2. `async` : Recompute the search index asynchronously as soon as possible with respect to a transacted write.
3. Scheduled: Use a cron expression to specify when the search index should be updated.

This is specified when creating a database by setting the property `search.reindex.mode` to `sync` , `async` , or to a valid cron expression. The default is `sync` . This behavior can be configured by the `search.reindex.mode` database option.

ACID Transactions

What follows is specific guidance with respect to Stardog's transactional semantics and guarantees.[14]

Atomicity

Databases may provide a guarantee of atomicity—groups of database actions (i.e., mutations) are irreducible and indivisible: either all of the changes happen or none of them happens. Stardog's transacted writes are atomic. Stardog does not support nested transactions.[15]

Consistency

Data stored should be valid with respect to the data model (in this case, RDF) and to the guarantees offered by the database, as well as to any application-specific integrity constraints that may exist. Stardog's transactions are guaranteed not to violate integrity constraints during execution. A transaction that would leave a database in an inconsistent or invalid state is aborted.

See the Validating Constraints section for a more detailed consideration of Stardog's integrity constraint mechanism.

Isolation

A Stardog connection will run in `READ COMMITTED` isolation level if it has not started an explicit transaction and will run in `READ COMMITTED SNAPSHOT` isolation level if it has started a transaction. In either mode, uncommitted changes will only be visible to the connection that made the changes: no other connection can see those values before they are committed. Thus, "dirty reads" can never occur. Neither mode locks the database; if there are conflicting changes, the latest commit wins.[16]

The difference between `READ COMMITTED` and `READ COMMITTED SNAPSHOT` isolation levels is that in the former case a connection will see updates committed by another connection immediately, whereas in the latter case a connection will see a transactionally consistent snapshot of the data as it existed at the start of the transaction and will not see any updates.

We illustrate the difference between these two levels with the following example where initially the database contains a single triple `:x :value 1`.

3. Table of the difference between RCI and RCSI

Time	Connection 1	Connection 2	Connection 3
0	`SELECT ?val {?x :val ?val}` ⇐ 1	`SELECT ?val {?x :val ?val}` ⇐ 1	`SELECT ?val {?x :val ?val}` ⇐ 1
1	BEGIN TX		

2	INSERT {:x :value 2}
	DELETE {:x :value ?old}		

3	`SELECT ?val {?x :val ?val}` ⇐ `2`	`SELECT ?val {?x :val ?val}` ⇐ `1`	`SELECT ?val {?x :val ?val}` ⇐ `1`
4			BEGIN TX
5	COMMIT		

6	`SELECT ?val {?x :val ?val}` ⇐ `2`	`SELECT ?val {?x :val ?val}` ⇐ `2`	`SELECT ?val {?x :val ?val}` ⇐ `1`

8			`INSERT {:x :value 3}`
			`DELETE {:x :value ?old}`

9			`COMMIT`
10	`SELECT ?val {?x :val ?val}` ⇐ `3`	`SELECT ?val {?x :val ?val}` ⇐ `3`	`SELECT ?val {?x :val ?val}` ⇐ `3`

Durability

By default Stardog's transacted writes are not durable; in some applications transactional durability is required and, thus, should be enabled.

Commit Failure Autorecovery

Stardog's transaction framework is largely maintenance free; but there are some rare conditions in which manual intervention may be needed.

Stardog's strategy for recovering automatically from (the very unlikely event of) commit failure is as follows:

1. Stardog will roll back the transaction upon a commit failure;
2. Stardog takes the affected database offline for maintenance;[17] then
3. Stardog will begin recovery, bringing the recovered database back online once that task is successful so that operations may resume.

With an appropriate logging configuration for production usage (at least error-level logging), log messages for the preceding recovery operations will occur. If for whatever reason the database fails to be returned automatically to online status, an administrator may use the CLI tools (i.e., `stardog-admin db online`) to attempt to online the database.

Optimizing Bulk Data Loading

Stardog tries hard to do bulk loading at database creation time in the most efficient and scalable way possible. But data loading time can vary widely, depending on factors in the data to be loaded, including the number of unique resources, etc. Here are some tuning tips that may work for you:

1. Load compressed data since compression minimizes disk access
2. Use a multicore machine since bulk loading is highly parallelized and indexes are built concurrently
3. Load multiple files together at creation time since different files will be parsed and processed concurrently improving the load speed
4. Turn off strict parsing (see Configuring a Database for the details).
5. If you are not using named graphs, use triples only indexing.

Capacity Planning

The primary system resources used by Stardog are CPU, memory, and disk.[18] Stardog will take advantage of multiple CPUs, cores, and core-based threads in data loading and in throughput-heavy or multi-user loads. And obviously Stardog performance is influenced by the speed of CPUs and cores. But some workloads are bound by main memory or by disk I/O (or both) more than by CPU. In general, use the fastest CPUs you can afford with the largest secondary caches and the most number of cores and core-based threads of execution, especially in multi-user workloads.

The following subsections provides more detailed guidance for the memory and disk resource requirements of Stardog.

Memory usage

Stardog uses system memory aggressively and the total system memory available to Stardog is often the most important factor in performance. Stardog uses both JVM memory (heap memory) and also the operating system memory outside the JVM (off heap memory). Having more system memory available is always good; however, increasing JVM memory too close to total system memory is not usually prudent as it may tend to increase Garbage Collection (GC) time in the JVM.

The following table shows recommended JVM memory and system memory requirements for Stardog.[19]

4. Table of Memory Usage for Capacity Planning

# of Triples	JVM Memory	Off-heap memory
100 million	3GB	3GB
1 billion	4GB	8GB
10 billion	8GB	64GB
20 billion	16GB	128GB
50 billion	16GB	256GB

Out of the box, Stardog CLI sets the maximum JVM memory to 2GB. This setting works fine for most small databases (up to, say, 100 million triples). As the database size increases, we recommend increasing JVM memory. You can increase the JVM memory for Stardog by setting the system property `STARDOG_JAVA_ARGS` using the standard JVM options. For example, you can set this property to `"-Xms4g -Xmx4g -XX:MaxDirectMemorySize=8g"` to increase the JVM memory to 4GB and off-heap to 8GB. We recommend setting the minimum heap size (`-Xms` option) as close to the max heap size (`-Xmx` option) as possible.

System Memory and JVM Memory

Stardog uses an off-heap, custom memory allocation scheme. Please note that the memory provisioning recommendations above are for two kinds of memory allocations for the JVM in which Stardog will run. The first is for memory that the JVM will manage explicitly (i.e., "JVM memory"); and the second, i.e., "Off-heap memory" is for memory that Stardog will manage explicitly, i.e., off the JVM heap, but for which the JVM should be notified via the `MaxDirectMemorySize` property. In most cases, this should be somewhat less than the total memory available to the underlying operating system as requirements dictate.

Disk usage

Stardog stores data on disk in a compressed format. The disk space needed for a database depends on many factors besides the number of triples, including the number of unique resources and literals in the data, average length of resource identifiers and literals, and how much the data is compressed. The following table shows typical disk space used by a Stardog database.

5. Table of Typical Disk Space Requirements

# of triples	Disk space
1 billion	70GB to 100GB
10 billion	700GB to 1TB

These numbers are given for information purposes only; the actual disk usage for a database may be significantly different in practice. Also it is important to note that the amount of disk space needed at creation time for bulk loading data is higher as temporary files will be created. The additional disk space needed at bulk loading time can be 40% to 70% of the final database size.

Disk space used by a database is non-trivially smaller if triples-only indexing is used. Triples-only indexing reduces overall disk space used by 25% on average; however, note the tradeoff: SPARQL queries involving named graphs perform significantly better with quads indexing.

The disk space used by Stardog is additive for multiple databases and there is very little disk space used other than what is required for the databases. To calculate the total disk space needed for multiple databases, one may sum the disk space needed by each database.

Using Stardog on Windows

Stardog provides batch (`.bat`) files for use on Windows; they provide roughly the same set of functionality provided by the Bash scripts which are used on Unix-like systems. There are, however, a few small differences between the two. When you start a server with `server start` on Windows, this does not detach to the background, it will run in the current console.

To shut down the server correctly, you should either issue a `server stop` command from another window or press `Ctrl` + `C` (and then `Y` when asked to terminate the batch job). Do not under any circumstance close the window without shutting down the server. This will simply kill the process without shutting down Stardog, which could cause your database to be corrupted.

The `.bat` scripts for Windows support our standard `STARDOG_HOME` and `STARDOG_JAVA_ARGS` environment variables which can be used to control where Stardog's database is stored and, usually, how much memory is given to Stardog's JVM when it starts. By default, the script will use the JVM that is available in the directory from which Stardog is run via the `JAVA_HOME` environment variable. If this is not set, it will simply execute `java` from within that directory.

Running Stardog as a Windows Service

You can run Stardog as a Windows Service using the following configuration. Please, note, that the following assumes commands are executed from a Command Prompt with administrative privileges.

Installing the Service

Change the directory to the Stardog installation directory:

```
cd c:\stardog-$VERSION
```

Configuring the Service

The default settings with which the service will be installed are

- 2048 MB of RAM
- `STARDOG_HOME` is the same as the installation directory
- the name of the installed service will be "Stardog Service"
- Stardog service will write logs to the "logs" directory within the installation directory

To change these settings, set appropriate environment variables:

- `STARDOG_MEMORY` : the amount of memory in MB (e.g., set `STARDOG_MEMORY` =4096)
- `STARDOG_HOME` : the path to `STARDOG_HOME` (e.g., set `STARDOG_HOME` =c:\\stardog-home)
- `STARDOG_SERVICE_DISPLAY_NAME` : a different name to be displayed in the list of services (e.g., set `STARDOG_SERVICE_DISPLAY_NAME` =Stardog Service)
- `STARDOG_LOG_PATH` : a path to a directory where the log files should be written (e.g., set `STARDOG_LOG_PATH` =c:\\stardog-logs)

If you have changed the default administrator password, you also need to modify `stop-service.bat` and specify the new username and password there (by passing `-u` and `-p` parameters in the line that invokes `stardog-admin server stop`).

Installing Stardog as a Service

Run the `install-service.bat` script.

At this point the service has been installed, but it is not running. To run it, see the next section or use any Windows mechanism for controlling the services (e.g., type `services.msc` on the command line).

Starting, Stopping, & Changing Service Configuration

Once the service has been installed, execute `stardog-serverw.exe` , which will allow you to configure the service (e.g., set whether the service is started automatically or manually), manually start and stop the service, as well as to configure most of the service parameters.

Uninstalling the Stardog Service

The service can be uninstalled by running `uninstall-service.bat` script.

Security

Stardog's security model is based on standard role-based access control: users have **permissions** over **resources** during **sessions**; permissions can be grouped into **roles**; and roles can be assigned to **users**.

Stardog uses Apache Shiro for authentication, authorization, and session management and jBCrypt for password hashing.

Resources

A resource is some Stardog entity or service to which access is controlled. Resources are identified by their **type** and their **name**. A particular resource is denoted as `type_prefix:name`. The valid resource types with their prefixes are shown below.

6. Table of System Resources

Resource	Prefix	Description
User	`user`	A user (e.g., `user:admin`)
Role	`role`	A role assigned to a user (`role:reader`)
Database	`db`	A database (`db:myDB`)
Named Graph	`named-graph`	A named graph (graph subset) (`named-`

Resource	Prefix	Description
		`graph:myDb\named-graph-id`)
Database Metadata	`metadata`	Metadata of a database (`metadata:myDB`)
Database Admin	`admin`	Database admin tasks (e.g., `admin:myDB`)
Integrity Constraints	`icv-constraints`	Integrity constraints associated with a database (e.g., `icv-constraints:myDB`)

Permissions

Permissions are composed of a **permission subject**, an **action**, and a **permission object**, which is interpreted as **the subject resource can perform the specified action over the object resource**.

Permission subjects can be of type `user` or `role` only. Permission objects can be of any valid type.

> **NOTE** `write` permission in Stardog refers to graph contents, including mutative operations performed via SPARQL Update (i.e., `INSERT`, `DELETE`, etc.). The other permissions, i.e., `create` and `delete`, apply to resources of the system itself, i.e., users, databases, database metadata, etc.

Valid actions include the following:

`read`

Permits reading the resource properties

`write`

Permits changing the resource properties

`create`

Permits creating new resources

`delete`

Permits deleting a resource

`grant`

Permits granting permissions over a resource

`revoke`

Permits revoking permissions over a resource

`execute`

Permits executing administration actions over a database

`all`

Special action type that permits all previous actions over a resource

Wildcards

Stardog understands the use of wildcards to represent sets of resources. A wildcard is denoted with the character `*`. Wildcards can be used to create complex permissions; for instance, we can give a user the ability to create any database by granting it a `create` permission over `db:*`. Similarly, wildcards can be used in order to revoke multiple permissions simultaneously.

Superusers

It is possible at user-creation time to specify that a given user is a superuser. Being a superuser is equivalent to having been granted an `all` permission over every resource, i.e., `*:*`. Therefore, as expected, superusers are allowed to perform any valid action over any existing (or future) resource.

Database Owner Default Permissions

When a user creates a resource, it is automatically granted `delete` , `write` , `read` , `grant` , and `revoke` permissions over the new resource. If the new resource is a database, then the user is additionally granted `write` , `read` , `grant` , and `revoke` permissions over `icv-constraints:theDatabase` and `execute` permission over `admin:theDatabase` . These latter two permissions give the owner of the database the ability to administer the ICV constraints for the database and to administer the database itself.

Default Security Configuration

> **WARNING** Out of the box, the Stardog security setup is minimal and **insecure**: `user:admin` with password set to "admin" is a superuser; `user:anonymous` with password "anonymous" has the "reader" role; `role:reader` allows `read` of any resource.

Do not deploy Stardog in production or in hostile environments with the default security settings.

Setting Password Constraints

To setup the constraints used to validate passwords when adding new users, configure the following settings in the `stardog.properties` configuration file.

- `password.length.min` : Sets the password policy for the minimum length of user passwords, the value can’t be less than 1 or greater than `password.length.max` . Default: `4` .
- `password.length.max` : Sets the password policy for the maximum length of user passwords, the value can’t be greater than 1024 or less than 1. Default: `20` .

- `password.regex`: Sets the password policy of accepted chars in user passwords, via a Java regular expression. Default: `[\\w@#$%]+`

Using a Password File

To avoid putting passwords into scripts or environment variables, you can put them into a suitably secured password file. If no credentials are passed explicitly in CLI invocations, or you do not ask Stardog to prompt you for credentials interactively, then it will look for credentials in a password file.

On a Unix system, Stardog will look for a file called `.sdpass` in the home directory of the user Stardog is running as; on a Windows system, it will look for `sdpass.conf` in `Application Data\stardog` in the home directory of the user Stardog is running as. If the file is not found in these locations, Stardog will look in the location provided by the `stardog.passwd.file` system property.

Password File Format

The format of the password file is as follows:

- any line that starts with a `#` is ignored
- each line contains a single password in the format: `hostname:port:database:username:password`.
- wildcards, `*`, are permitted for any field but the password field; colons and backslashes in fields are escaped with `\`.

For example,

```
#this is my password file; there are no others like it and this one is mine anyway...
*:*:*:flannery:aNahthu8
*:*:summercamp:jemima:foh9Moaz
```

Of course you should secure this file carefully, making sure that only the user that Stardog runs as can read it.

Named Graph Security

Stardog's security model is based on standard RBAC notions: users have permissions over resources during sessions; permissions can be grouped into roles; and roles can be assigned to users. Stardog defines a database resource type so that users and roles can be given read or write access to a database. With Named Graph Security added in Stardog 3.1, Stardog lets you specify which named graphs a user can read from or write to; that is, named graphs are now an explicit resource type in Stardog's security model.

Example

To grant a user permissions to a named graph,

```
$ stardog-admin user grant -a read -o named-graph:myDB\http://example.org/g1 myUser
$ stardog-admin user grant -a write -o named-graph:myDB\http://example.org/g2 myUser
```

Note the use of "\" to separate the name of the database ("myDB") from the named graph identifier ("http://example.org/g1").

IMPORTANT

Named Graph Security is **disabled** by default (for backwards compatibility with the installed base). It can be enabled globally (or per database) by setting `security.named.graphs=true`, in `stardog.properties` globally, or per database.

Named Graph Operations

Stardog does not support the notion of an empty named graph; thus, there is no operation to create a named graph. Deleting a named graph is simply removing all the triples in that named graph; so it's also not a special operation. For this reason, only **read** and **write** permissions can be used with named graphs and **create** and **delete** permissions cannot be used with named graphs.

How Named Graph Permissions Work

The set of named graphs to which a user has read or write access is **the union of named graphs for which it has been given explicit access plus the named graphs for which the user's roles have been given access**.

Querying

An effect of named graph permissions is changing the RDF Dataset associated with a query. The default and named graphs specified for an RDF Dataset will be filtered to match the named graphs that a user has read access to.

> **NOTE** A read query never triggers a security exception due to named graph permissions. The graphs that a user cannot read from would be **silently dropped from the RDF dataset for the query**, which may cause the query to return no answers, despite there being matching triples in the database.

The RDF dataset for SPARQL update queries will be modified similarly based on read permissions.

> **NOTE** The RDF dataset for an update query affects only the `WHERE` clause.

Writing

Write permissions are enforced by throwing a security exception whenever a named graph is being updated by a user that does not have write access to the graph. Adding a triple to an unauthorized named graph will raise an exception even if that triple already exists in the named graph. Similarly trying to remove a non-existent triple from an unauthorized graph raises an error.

> **NOTE** The unauthorized graph may not exist in the database; any graph that is not explicitly listed in a user's permissions is unauthorized.

Updates either succeed as a whole or fail. If an update request tries to modify both an authorized graph an unauthorized graph, it would fail without making any modifications.

Reasoning

Stardog allows a set of named graphs to be used as the schema for reasoning. The OWL axioms and rules defined in these graphs are extracted and used in the reasoning process. The schema graphs are specified in the database configuration and affect all users running reasoning queries.

Named graph permissions do **not** affect the schema axioms used in reasoning and every reasoning query will use the same schema axioms even though some users might **not** have been granted explicit read access to schema graphs. But non-schema axioms in those named graphs would **not** be visible to users without authorization.

Enterprise Authentication

Stardog can use an LDAP server to authenticate enterprise users. Stardog assumes the existence of two different groups to identify regular and superusers, respectively. Groups must be identified with the `cn` attribute and be instances of the `groupOfNames` object class. Users must be specified using the `member` attribute.

For example,

```
dn: cn=stardogSuperUsers,ou=group,dc=example,dc=com
cn: stardogSuperUsers
objectclass: groupOfNames
member: uid=superuser,ou=people,dc=example,dc=com

dn: cn=stardogUsers,ou=group,dc=example,dc=com
cn: stardogUsers
objectclass: groupOfNames
member: uid=regularuser,ou=people,dc=example,dc=com
member: uid=anotherregularuser,ou=people,dc=example,dc=com
```

Credentials and other user information are stored as usual:

```
dn: uid=superuser,ou=people,dc=example,dc=com
objectClass: inetOrgPerson
```

```
cn: superuser
sn: superuser
uid: superuser
userPassword: superpassword
```

Configuring Stardog

In order to enable LDAP authentication in Stardog, we need to include the following **mandatory** properties in `stardog.properties`:

- `security.realms`: with a value of `ldap`
- `ldap.provider.url`: The URL of the LDAP server
- `ldap.security.principal`: An LDAP user allowed to retrieve group members from the LDAP server
- `ldap.security.credentials`: The principal’s password
- `ldap.user.dn.template`: A template to form LDAP names from Stardog usernames
- `ldap.group.lookup.string`: A string to lookup the Stardog user groups
- `ldap.users.cn`: The `cn` of the group identifying regular Stardog users
- `ldap.superusers.cn`: The `cn` of the group identifying Stardog super users

Here’s another example:

```
security.realms = ldap
ldap.provider.url = ldap://localhost:5860
ldap.security.principal = uid=admin,ou=people,dc=example,dc=com
ldap.security.credentials = secret
ldap.user.dn.template = uid={0},ou=people,dc=example,dc=com
ldap.group.lookup.string = ou=group,dc=example,dc=com
ldap.users.cn = stardogUsers
ldap.superusers.cn = stardogSuperUsers
```

User Management

Users can no longer be added/removed/modified via Stardog. User management is delegated to the LDAP server.

An LDAP Quirk

When Stardog manages users, instead of delegating to LDAP, when a user is created, they are assigned the permission `read:user:$NEW_USER`. But when user management is delegated to LDAP, this permission is not automatically created at new user creation time in Stardog and, therefore, it should be added manually to Stardog. If this doesn't happen, users won't be able to—among other things—log into the Web Console.

User Cache

Stardog does not cache the list of users. The LDAP server is contacted every time a user authenticates, which ensures that users deleted from LDAP will not be able to authenticate in Stardog.

Authorization

The LDAP server is used for **authentication only**. Permissions and roles are assigned in Stardog.

Stale Permissions/Roles

Permissions and roles in Stardog might refer to no longer existing users (those who were deleted from the LDAP server). This is safe as these users will not be able to authenticate (see above).

It is possible to configure Stardog to periodically clean up the list of permissions and roles according to the latest users in the LDAP server. In order to do this, we pass a Quartz cron expression using the `ldap.consistency.scheduler.expression` property:

```
## Execute the consistency cleanup at 6pm every day
ldap.consistency.scheduler.expression = 0 0 18 * * ?
```

Managing Stardog Securely

Stardog resources can be managed securely by using the tools included in the admin CLI or by programming against Stardog APIs. In this section we describe the permissions required to manage various Stardog resources either by CLI or API.

Users

Create a user

`create` permission over `user:*`. Only superusers can create other superusers.

Delete a user

`delete` permission over the user.

Enable/Disable a user

User must be a superuser.

Change password of a user

User must be a superuser or user must be trying to change its own password.

Check if a user is a superuser

`read` permission over the user or user must be trying to get its own info.

Check if a user is enabled

`read` permission over the user or user must be trying to get its own info.

List users

Superusers can see all users. Other users can see only users over which they have a permission.

Roles

Create a role

`create` permission over `role:*`.

Delete a role

`delete` permission over the role.

Assign a role to a user

`grant` permission over the role **and** user must have all the permissions associated to the role.

Unassign a role from a user

`revoke` permission over the role **and** user must have all the permissions associated to the role.

List roles

Superusers can see all roles. Other users can see only roles they have been assigned or over which they have a permission.

Databases

Create a database

`create` permission over `db:*`.

Delete a database

`delete` permission over `db:theDatabase`.

Add/Remove integrity constraints to a database

`write` permission over `icv-constraints:theDatabase`.

Verify a database is valid

`read` permission over `icv-constraints:theDatabase`.

Online/Offline a database

`execute` permission over `admin:theDatabase`.

Migrate a database

`execute` permission over `admin:theDatabase`.

Optimize a database

`execute` permission over `admin:theDatabase`.

List databases

Superusers can see all databases. Regular users can see only databases over which they have a permission.

Permissions

Grant a permission

`grant` permission over the permission object **and** user must have the permission that it is trying to grant.

Revoke a permission from a user or role over an object resource

`revoke` permission over the permission object **and** user must have the permission that it is trying to revoke.

List user permissions

User must be a superuser or user must be trying to get its own info.

List role permissions

User must be a superuser or user must have been assigned the role.

Deploying Stardog Securely

To ensure that Stardog's RBAC access control implementation will be effective, all non-administrator access to Stardog databases should occur over network (i.e., non-native) database connections.[20]

To ensure the confidentiality of user authentication credentials when using remote connections, the Stardog server should only accept connections that are encrypted with SSL.

Configuring Stardog to use SSL

Stardog HTTP server includes native support for SSL. The SNARL server (via `snarls://`) also supports SSL. To enable Stardog to optionally support SSL connections, just pass `--enable-ssl` to the server start command. If you want to require the server to use SSL only, that is, to reject any non-SSL connections, then use `--require-ssl`.

When starting from the command line, Stardog will use the standard Java properties for specifying keystore information:

- `javax.net.ssl.keyStorePassword` (the password)
- `javax.net.ssl.keyStore` (location of the keystore)
- `javax.net.ssl.keyStoreType` (type of keystore, defaults to JKS)

These properties are checked first in `stardog.properties`; then in JVM args passed in from the command line, e.g. `-Djavax.net.ssl.keyStorePassword=mypwd`. If you're creating a Server progammatically via `ServerBuilder`, you can specify values for these properties using the appropriate `ServerOptions` when creating the server. These values will override anything specified in `stardog.properties` or via normal JVM args.

Configuring Stardog Client to use SSL

Stardog HTTP client supports SSL when the `https:` scheme is used in the database connection string; likewise, it uses SSL for SNARL when the connection string uses the `snarls:` scheme. For example, the following invocation of the Stardog command line utility will initiate an SSL connection to a remote database:

```
$ stardog status -c https://stardog.example.org/sp2b_10k
```

If the client is unable to authenticate to the server, then the connection will fail and an error message like the following will be generated.

```
Error during connect.  Cause was SSLPeerUnverifiedException: peer not authenticated
```

The most common cause of this error is that the server presented a certificate that was not issued by an authority that the client trusts. The Stardog HTTP client driver uses standard Java security components to access a store of trusted certificates. By default, it trusts a list of certificates installed with the Java runtime environment, but it can be configured to use a custom trust store.[21]

The client driver can be directed to use a specific Java KeyStore file as a trust store by setting the `javax.net.ssl.trustStore` system property. To address the authentication error above, that trust store should contain the issuer of the server's certificate. Standard Java tools can create such a file. The following invocation of the `keytool` utility creates a new trust store named `my-truststore.jks` and initializes it with the certificate in `my-trusted-server.crt`. The tool will prompt for a passphrase to associate with the trust store. This is not used to encrypt its contents, but can be used to ensure its integrity.[22]

```
$ keytool -importcert  -keystore my-truststore.jks -alias stardog-server -file
my-trusted-server.crt
```

The following Stardog command line invocation uses the newly created truststore.

```
$ STARDOG_JAVA_ARGS=”-Djavax.net.ssl.trustStore=my-truststore.jks” stardog \
status -c https://stardog.example.org/sp2b_10k
```

For custom Java applications that use the Stardog HTTP client driver, the system property can be set programmatically or when the JVM is initialized.

The most common deployment approach requiring a custom trust store is when a self-signed certificate is presented by the Stardog server. For connections to succeed, the Stardog client must trust the self-signed certificate. To accomplish this with the examples given above, the self-signed certificate should be in the `my-trusted-server.crt` file in the keytool invocation.

A client may also fail to authenticate to the server if the hostname in the Stardog database connection string does not match a name contained in the server certificate.[23]

This will cause an error message like the following

```
Error during connect.  Cause was SSLException: hostname in certificate didn't match
```

The client driver does not support connecting when there's a mismatch; therefore, the only workarounds are to replace the server's certificate or modify the connection string to use an alias for the same server that matches the certificate.

High Availability Cluster

In this section we explain how to configure, use, and administer Stardog Cluster for uninterrupted operations.

Stardog Cluster is a collection of Stardog Server instances running on one or more virtual or physical machines that, **from the client's perspective**, behave like a single Stardog Server instance. To fully achieve this effect requires DNS (i.e., with `SRV` records) and proxy configuration that's left as an exercise for the user. Of course Stardog Cluster should have some different operational properties, the main one of which is high availability. But from the client's perspective Stardog Cluster should be indistinguishable from non-clustered Stardog.[24]

> **NOTE** High Availability requires **at least** three nodes in the Cluster. Stardog Cluster works best, with respect to fault resiliency, with a cluster size that is an odd-number greater than or equal to three : 3, 5, 7, etc.[25] With respect to performance, larger cluster sizes perform better than smaller ones.

Cluster Guarantees

Stardog Cluster implements an atomic commitment protocol based on two-phase commit (2PC) over a shared replicated memory that's provided by Apache ZooKeeper. A cluster is composed of a set of Stardog servers running together. One of the servers is known as the Coordinator and the rest as Participants.

In case the Coordinator fails at any point, a new Coordinator will be elected out of the remaining available Participants. Stardog Cluster supports both `read` (e.g., querying) and `write` (e.g., adding data) requests. Read requests are load-balanced over the available Participants, whereas write requests are transparently forwarded to and handled by the Coordinator. In some future release we may change the protocol implemented by the Cluster and thus change some of the allowable topologies, including multiple-writers and multiple-readers.

When a client commits a transaction (containing a list of `write` requests), it will be acknowledged by the Coordinator only **after every non-failing Participant has committed the transaction**. If a Participant fails during the process of committing a transaction, it will be expelled from the cluster by the Coordinator and put in a temporary `failed` state.

If the Coordinator fails during the process, the transaction will be aborted, and a new Coordinator will be elected automatically. Since `failed` nodes are not used for any subsequent `read` or `write` requests, **if a commit is acknowledged by the Coordinator, then Stardog Cluster guarantees that the data has been accordingly modified at every available node in the cluster**.

While this approach is less performant with respect to write operations than eventual consistency used by other distributed databases, typically those databases offer a much less expressive data model than Stardog, which makes an eventually consistency model more appropriate for those systems. But since Stardog’s data model is not only richly expressive but rests in part on provably correct semantics, we think that a strong consistency model is worth the cost.[26]

Configuration

To deploy Stardog Cluster you use `stardog-admin` commands and some additional configuration. Stardog Cluster depends on Apache ZooKeeper.

In Stardog 3.0 we added `stardog-admin cluster generate` to bootstrap a cluster configuration and, thus, to ease installation. You may be able to get away with simply passing a list of hostnames or IP addresses for the cluster’s nodes.

```
$ stardog-admin cluster generate --output-dir /home/stardog 10.0.0.1 10.0.0.2 10.0.0.3
```

See the man page for the details.

In the following installation notes, we install Stardog Cluster in a 3-node configuration. If you need larger cluster, adjust accordingly.[27]

1. Install Stardog 3.1.4 on each machine in the cluster.

   NOTE

   > The smart thing to do here, of course, is to use whatever infrastructure you have in place to automate software installation. Adapting Stardog installation to Chef, Puppet, cfengine, etc. is left as an exercise for the reader.

2. Make sure a valid Stardog license key (whether Developer, Enterprise, or a 30-day eval key) for the size of cluster you’re creating exists and resides in `STARDOG_HOME` on each node. You must also have a `stardog.properties` file with the following information **for each node in the cluster**:

   ```
   # Flag to enable the cluster, without this flag set, the rest of the properties
   have no effect
   pack.enabled=true
   # this node's IP address (or hostname), and port where other Stardog nodes are
   going to connect
   pack.node.address=196.69.68.1
   # the connection string for ZooKeeper where cluster state is stored
   pack.cluster.address=196.69.68.1:2180,196.69.68.2:2180,196.69.68.3:2180
   # credentials used for securing ZooKeeper state
   pack.cluster.username=pack
   pack.cluster.password=admin
   ```

   `pack.cluster.address` is a ZooKeeper connection string where cluster stores its state. `pack.cluster.username` and `pack.cluster.password` are user and password tokens for ZooKeeper cluster communications and may be different from actual Stardog users and passwords; however, **all nodes must use the same user and password combination.** `pack.node.address` is not a required property. The local address of the node, by default, is `InetAddress.getLocalhost().getAddress()`, which should work for many deployments. However if you’re using an atypical network topology and the default value is not correct, you can provide a value for this property.

3. Create the ZooKeeper configuration for each node. This config file is just a standard ZooKeeper configuration file. The following config file should be sufficient for most cases.

   On node 1:

   ```
   tickTime=2000
   # Make sure this directory exists and
   # ZK can write and read to and from it.
   dataDir=/data/zookeeperdata/
   clientPort=2180
   initLimit=5
   syncLimit=2
   # This is an enumeration of all nodes in
   # the cluster and must be identical in
   # each node's config.
   server.1=196.69.68.1:2888:3888
   server.2=196.69.68.2:2888:3888
   server.3=196.69.68.3:2888:3888
   ```

   On node 2:

   ```
   tickTime=2000
   dataDir=/data/zookeeperdata/
   clientPort=2180
   initLimit=5
   syncLimit=2
   server.1=196.69.68.1:2888:3888
   server.2=196.69.68.2:2888:3888
   server.3=196.69.68.3:2888:3888
   ```

   Finally, on node 3:

   ```
   tickTime=2000
   dataDir=/data/zookeeperdata/
   clientPort=2180
   initLimit=5
   syncLimit=2
   server.1=196.69.68.1:2888:3888
   ```

```
server.2=196.69.68.2:2888:3888
server.3=196.69.68.3:2888:3888
```

NOTE The `clientPort` specified in `zookeeper.properties` and the ports used in `pack.cluster.address` in `stardog.properties` must be the same.

4. `dataDir` is where ZooKeeper persists cluster state and where it writes log information about the cluster.

```
$ mkdir /data/zookeeperdata # on node 1
$ mkdir /data/zookeeperdata # on node 2
$ mkdir /data/zookeeperdata # on node 3
```

5. ZooKeeper requires a `myid` file in the `dataDir` folder to identify itself, you will create that file as follows for `node1` and `node2`, respectively:

```
$ echo 1 > /data/zookeeperdata/myid # on node 1
$ echo 2 > /data/zookeeperdata/myid # on node 2
$ echo 3 > /data/zookeeperdata/myid # on node 3
```

Installation

In the next few steps you will use the Stardog Admin CLI commands to deploy Stardog Cluster: that is, ZooKeeper, the Proxy, and Stardog itself.

1. To start ZooKeeper’s part of Cluster, use the `stadog-admin cluster` subcommand:

```
$ ./stardog-admin cluster zkstart --home ~/stardog # on node 1
$ ./stardog-admin cluster zkstart --home ~/stardog # on node 2
$ ./stardog-admin cluster zkstart --home ~/stardog # on node 3
```

Which uses the `zookeeper.properties` config file in `~/stardog` and log its output to `~/stardog/zookeeper.log`. If your `$STARDOG_HOME` is set to `~/stardog`, then you don't need to specify the `--home` option. For more info about the command:

```
$ ./stardog-admin help cluster zkstart
```

Once ZooKeeper is started, you can start Stardog Cluster:

```
$ ./stardog-admin server start --home ~/stardog --port 5821 # on node 1
$ ./stardog-admin server start --home ~/stardog --port 5821 # on node 2
$ ./stardog-admin server start --home ~/stardog --port 5821 # on node 3
```

Again, if your `$STARDOG_HOME` is set to `~/stardog`, you don't need to specify the `--home` option.

NOTE We start Stardog here on the non-default port (`5821`) so that the Proxy can run on the default port (`5820`), which means that Stardog clients can act normally (i.e., use the default port, `5820`) since they need to interact with the Proxy.

2. Start the Stardog Cluster Proxy:

```
$ ./stardog-admin cluster proxystart --zkconnstr
196.69.68.1:2180,196.69.68.2:2180,196.69.68.3:2180 \
        --user pack --password admin --port 5820 # on node 1
$ ./stardog-admin cluster proxystart --zkconnstr
196.69.68.1:2180,196.69.68.2:2180,196.69.68.3:2180 \
   --user pack --password admin --port 5820 # on node 2
$ ./stardog-admin cluster proxystart --zkconnstr
196.69.68.1:2180,196.69.68.2:2180,196.69.68.3:2180 \
        --user pack --password admin --port 5820 # on node 3
```

Note that the `zkconnstr` option is the same connection string as `pack.cluster.address` in `stardog.properties`, and `user` and `password` are the same as `pack.cluster.username` and `pack.cluster.password`, respectively. For more information on the proxy configuration execute:

```
$ ./stardog-admin help cluster proxystart
```

Now Stardog Cluster is running on 3 nodes, one each on 3 machines. Since the proxy was conveniently configured to use port `5820` you can execute standard Stardog CLI commands to the Cluster:

```
$ ./stardog-admin db create -n myDb
$ ./stardog data add myDb /path/to/my/data
$ ./stardog query myDb "select * { ?s ?p ?o } limit 5"
```

Running Stardog Cluster on a Single Machine

For testing, development, or other purposes it may be advantageous to run Stardog Cluster on a single machine. Of course this won't provide much actual high availability, but it may be a reasonable choice in some cases.

The following changes must be made to the multi-machine configuration and installation described previously:

Stardog configuration

- different `STARDOG_HOME` for each node
- different Stardog port for each node
  - the easiest setup is if no Stardog listens on port `5820`
- at least one (but no more than one) proxy on port `5820`
- different Watchdog port for each node via `watchdog.port` in `stardog.properties`

ZooKeeper configuration

- different, that is, non-overlapping port ranges for each node

- different data directory for each node (e.g., different subdirectories in `/data/zookeeperdata`)

Cluster Topologies & Cluster Size

In the configuration instructions above, we assume a particular Cluster typology, which is to say, for each node `n` of a cluster, we run Stardog, ZooKeeper, and a Proxy. But this is not the only typology supported by Stardog Cluster. ZooKeeper nodes run independently, so other typologies—three ZooKeeper servers and five Stardog servers are possible—you just have to point Stardog to the corresponding ZooKeeper cluster.

To add more Stardog Cluster nodes, simply repeat the steps for Stardog on additional machines. Generally, as mentioned above, Stardog Cluster size should be an odd number greater or equal to 3.

NOTE

ZooKeeper uses a very write heavy protocol; having Stardog and ZooKeeper both writing to the **same** disk can yield contention issues, resulting in timeouts at scale. We recommend at a minimum having the two services writing to separate disks to reduce contention or, ideally, have them run on separate nodes entirely.

Stardog Cluster Client

To use Stardog Cluster with standard Stardog clients and CLI tools in the ordinary way--`stardog-admin` and `stardog`--you must have Stardog installed locally. With the provided Stardog binaries in the Stardog Cluster distribution you can query the state of Cluster:[28]

```
$ ./stardog-admin --server snarl://<ipaddress>:5820/ cluster info
```

where `ipaddress` is the IP address of any of the nodes in the cluster. This will print the available nodes in the cluster, as well as the roles (participant or coordinator). You can also input the proxy IP address and port to get the same information.

To add or remove data, issue `stardog data add` or `remove` commands to any node in the cluster. Queries can be issued to any node in the cluster using the `stardog query` command. All the `stardog-admin` features are also available in Cluster, which means you can use any of the commands to create databases, adminster users, and the rest of the functionality.

Cluster Management

Stardog Cluster may be managed more easily by using Stardog Release, a new, open source Stardog Cluster management system. Stardog Release is based on BOSH, "an open source tool chain for release engineering, deploying and lifecycle management of large scale distributed services" from Pivotal Cloud Foundry. Using BOSH, you can manage Stardog Cluster more easily than ever before; with Stardog Release you can install and update Stardog Cluster "on large numbers of VMs over many IaaS platforms with the absolute minimum of configuration changes".

See the Stardog Release Github repo for installation, configuration, and usage.

See the BOSH docs for more details.

Limitations

The notable limitations of Stardog Release as of Stardog 3.1:

- supports AWS and BOSH Lite (local machine) deployments only
  - in other words, if you want OpenStack, vSphere, or vCloud deployment, ping us.
- VMs only

OWL & Rule Reasoning

In this chapter we describe how to use Stardog's reasoning capabilities; we address some common problems and known issues. We also describe Stardog's approach to query answering with reasoning in some detail, as well as a set of guidelines that contribute to efficient query answering with reasoning. If you are not familiar with the terminology, you can peruse the section on terminology.

The semantics of Stardog's reasoning is based in part on the OWL 2 Direct Semantics Entailment Regime. However, the implementation of Stardog's reasoning system is worth understanding as well. For the most part, **Stardog performs reasoning in a lazy and late-binding fashion: it does not materialize inferences; but, rather, reasoning is performed at query time according to a user-specified "reasoning type".** This approach allows for maximum flexibility[29] while maintaining excellent performance. The one exception to this general approach is equality reasoning which is eagerly materialized. See Same As Reasoning for more details.

Reasoning Types

In Stardog 3.0, reasoning can be enabled or disabled using a simple boolean flag—in HTTP, `reasoning`; in CLI, `-r` or `--reasoning`; in the Web Console, a reasoning button in the query panel; and in Java APIs, a connection option or a query option:

`false` No axioms or rules are considered; no reasoning is performed.

`true` Axioms and rules are considered and reasoning is performed according to the value of the `reasoning.type` database option.

Reasoning is disabled by default; that is, no reasoning is performed without explicitly setting the reasoning flag to "true".

When reasoning is enabled by the boolean flag, the axioms and rules in the database are first filtered according to the value of the `reasoning.type` database option. The default value of `reasoning.type` is `SL` and for the most part users don't need to worry too much about which reasoning type is necessary since `SL` covers all of the OWL 2 profiles as well as user-defined rules via SWRL. However, this value may be set to any other reasoning type that Stardog supports: `RDFS` is the OWL 2 axioms allowed in RDF Schema (mainly subclasses, subproperties, domain, and ranges); `QL` for the OWL 2 QL axioms; `RL` for the OWL 2 RL axioms; `EL` for the OWL 2 EL axioms; `DL` for OWL 2 DL axioms; and `SL` for a combination of RDFS, QL, RL, and EL axioms, plus SWRL rules. Any axiom outside the selected type will be ignored by the reasoner.

The `DL` reasoning type behaves significantly different than other types. Stardog normally uses the Query Rewriting technique for reasoning which scales very well with increasing number of instances; only the schema needs to be kept in memory. But query rewriting cannot handle axioms outside the OWL 2 profiles; however, `DL` reasoning type can be used so that no axiom or rule is ignored as long as they satisfy the OWL 2 DL restricions. With `DL` reasoning, both the schema and the instance data need to pulled into memory, which limits its applicability with large number of instances. `DL` reasoning also requires the database to be logically consistent or no reasoning can be be performed. Finally, `DL` reasoning requires more computation upfront compared to query rewriting which exhibits a "pay-as-you-go" behavior.

The `reasoning.type` can also be set to the special value `NONE` which will filter all axioms and rules thus effectively disables reasoning. This value can be used for the database option to prevent reasoning to be used by any client even though they might enable it with the boolean flag on the client side.

Using Reasoning

In order to perform query evaluation with reasoning, Stardog requires a schema[30] to be present in the database. Since schemas are serialized as RDF, they are loaded into a Stardog database in the same way that any RDF is loaded into a Stardog database. Also, note that, since the schema is just more RDF triples, it may change as needed: it is neither fixed nor compiled in any special way.

The schema may reside in the default graph, in a specific named graph, or in a collection of graphs. **You can tell Stardog where the schema is by setting the `reasoning.schema.graphs` property to one or more named graph URIs.** If you want the default graph to be considered part of the schema, then you can use the special built-in URI `tag:stardog:api:context:default`. If you want to use all named graphs (that is, to tell Stardog to look for the schema in every named graph), you can use `tag:stardog:api:context:all`.

NOTE

As of Stardog 3.0, **the default value for this property is to use all graphs, i.e., `tag:stardog:api:context:all`.**

This design is intended to support both of Stardog's primary use cases:

1. managing the data that constitutes the schema
2. reasoning with the schema during query evaluation

Query Answering

All of Stardog's interfaces (API, network, and CLI) support reasoning during query evaluation.

Command Line

In order to evaluate queries in Stardog using reasoning via the command line, we use the reasoning flag:

```
$ ./stardog query --reasoning myDB "SELECT ?s { ?s a :C } LIMIT 10"
```

HTTP

For HTTP, the reasoning flag is specified with the other HTTP request parameters:

```
$ curl -u admin:admin -X GET "http://localhost:5822/myDB/
query?reasoning=true&query=..."
```

Reasoning Connection API

In order to use the `ReasoningConnection` API one needs to enable reasoning. See the Java Programming section for details.

Currently, the API has two methods:

- `isConsistent()`, which can be used to check if the database is (logically) consistent with respect to the reasoning type.
- `isSatisfiable(URI theURIClass)`, which can be used to check if the given class if satisfiable with respect to the database and reasoning type.

Explaining Reasoning Results

Stardog can be used to check if the current datbase logically entails a set of triples; moreover, Stardog can explain why this is so.[31] An explanation of an inference is the minimum set of statements explicitly stored in the database that, together with the schema and any valid inferences, logically justify the inference. Explanations are useful for understanding data, schema, and their interactions, especially when large number of statements interact with each other to infer new statements.

Explanations can be retrieved using the CLI by providing an input file that contains the inferences to be explained:

```
$ stardog reasoning explain myDB inference_to_explain.ttl
```

The output is displayed in a concise syntax designed to be legible; but it can be rendered in any one of the supported RDF syntaxes if desired. Explanations are also accessible through the Stardog’s extended HTTP protocol and discussion of SNARL. See the examples included in the distribution for more details about retrieving explanations programmatically.

Proof Trees

Proof trees are a hierarchical presentation of multiple explanations (of inferences) to make data, schemas, and rules more intelligible. Proof trees[32] provide an explanation for an inference or an inconsistency as a **hierarchical structure**. Nodes in the proof tree may represent an assertion in a Stardog database. Multiple assertion nodes are grouped under an inferred node.

Example

For example, if we are explaining the inferred triple `:Alice rdf:type :Employee`, the root of the proof tree will show that inference:

```
INFERRED :Alice rdf:type :Employee
```

The children of an inferred node will provide more explanation for that inference:

```
INFERRED :Alice rdf:type :Employee
    ASSERTED :Manager rdfs:subClassOf :Employee
    INFERRED :Alice rdf:type :Manager
```

The fully expanded proof tree will show the asserted triples and axioms for every inference:

```
INFERRED :Alice rdf:type :Employee
    ASSERTED :Manager rdfs:subClassOf :Employee
    INFERRED :Alice rdf:type :Manager
        ASSERTED :Alice :supervises :Bob
        ASSERTED :supervises rdfs:domain :Manager
```

The CLI explanation command prints the proof tree using indented text; but, using the SNARL API, it is easy to create a tree widget in a GUI to show the explanation tree, such that users can expand and collapse details in the explanation.

Another feature of proof trees is the ability to merge multiple explanations into a single proof tree with multiple branches when explanations have common statements. Consider the following example database:

```
#schema
:Manager rdfs:subClassOf :Employee
:ProjectManager rdfs:subClassOf :Manager
:ProjectManager owl:equivalentClass (:manages some :Project)
:supervises rdfs:domain :Manager
:ResearchProject rdfs:subClassOf :Project
:projectID rdfs:domain :Project

#instance data
:Alice :supervises :Bob
:Alice :manages :ProjectX
:ProjectX a :ResearchProject
:ProjectX :projectID "123-45-6789"
```

In this database, there are three different unique explanations for the inference `:Alice rdf:type :Employee`:

Explanation 1

```
:Manager rdfs:subClassOf :Employee
:ProjectManager rdfs:subClassOf :Manager
:supervises rdfs:domain :Manager
:Alice :supervises :Bob
```

Explanation 2

```
:Manager rdfs:subClassOf :Employee
:ProjectManager rdfs:subClassOf :Manager
:ProjectManager owl:equivalentClass (:manages some :Project)
:ResearchProject rdfs:subClassOf :Project
:Alice :manages :ProjectX
:ProjectX a :ResearchProject
```

Explanation 3

```
:Manager rdfs:subClassOf :Employee
:ProjectManager rdfs:subClassOf :Manager
:ProjectManager owl:equivalentClass (:manages some :Project)
:projectID rdfs:domain :Project
:Alice :manages :ProjectX
:ProjectX :projectID "123-45-6789"
```

All three explanations have some triples in common; but when explanations are retrieved separately, it is hard to see how these explanations are related. When explanations are merged, we get a single proof tree where alternatives for subtrees of the proof are shown inline. In indented text rendering, the merged tree for the above explanations would look as follows:

```
INFERRED :Alice a :Employee
   ASSERTED :Manager rdfs:subClassOf :Employee
   1.1) INFERRED :Alice a :Manager
      ASSERTED :supervises rdfs:domain :Manager
      ASSERTED :Alice :supervises :Bob
   1.2) INFERRED :Alice a :Manager
      ASSERTED :ProjectManager rdfs:subClassOf :Manager
      INFERRED :Alice a :ProjectManager
         ASSERTED :ProjectManager owl:equivalentClass (:manages some :Project)
         ASSERTED :Alice :manages :ProjectX
         2.1) INFERRED :ProjectX a :Project
            ASSERTED :projectID rdfs:domain :Project
            ASSERTED :ProjectX :projectID "123-45-6789"
         2.2) INFERRED :ProjectX a :Project
            ASSERTED :ResearchProject rdfs:subClassOf :Project
            ASSERTED :ProjectX a :ResearchProject
```

In the merged proof tree, alternatives for an explanation are shown with a number id. In the above tree, `:Alice a :Manager` is the first inference for which we have multiple explanations so it gets the id `1`. Then each alternative explanation gets an id appended to this (so explanations `1.1` and `1.2` are both alternative explanations for inference `1`). We also have multiple explanations for inference `:ProjectX a :Project` so its alternatives get ids `2.1` and `2.2`.

User-defined Rule Reasoning

Many reasoning problems may be solved with OWL's axiom-based approach; but, of course, not all reasoning problems are amenable to this approach. A user-defined rules approach complements the OWL axiom-based approach nicely and increases the expressive power of a reasoning system from the user's point of view. Many RDF databases support user-defined rules only. Stardog is the only RDF database that comprehensively supports both axioms and rules.

Some problems (and some people) are simply a better fit for a rules-based approach to modeling and reasoning than to an axioms-based approach (and, of course, *vice versa*).

NOTE There isn't a one-size-fits-all answer to the question "rules or axioms or both?" Use the thing that makes the most sense given the task at hand. This is engineering, not religion.

Stardog supports user-defined rule reasoning together with a rich set of built-in functions using the SWRL syntax and builtin-ins library. In order to apply SWRL user-defined rules, you must include the rules as part of the database's schema: that is, put your rules where your axioms are, i.e., in the schema. Once the rules are part of the schema, they will be used for reasoning automatically when using the **SL** reasoning type.

Assertions implied by the rules **will not** be materialized. Instead, rules are used to expand queries just as regular axioms are used.

NOTE To trigger rules to fire, execute a relevant query—simple and easy as the truth.

Stardog Rules Syntax

Stardog supports two different syntaxes for defining rules. The first is native Stardog Rules syntax and is based on SPARQL, so you can re-use what you already know about SPARQL to write rules. **Unless you have specific requirements otherwise, you should use this syntax for user-defined rules in Stardog.** The second is the **de facto** standard RDF/XML syntax for SWRL. It has the advantage of being supported in many tools; but it's not fun to read or to write. You probably don't want to use it. Better: don't use this syntax!

Stardog Rules Syntax is basically SPARQL "basic graph patterns" (BGPs) plus some very explicit new bits (`IF-THEN`) to denote the head and the body of a rule.[33] You define URI prefixes in the normal way (examples below) and use regular SPARQL variables for rule variables. As you can see, some SPARQL 1.1 syntactic sugar—property paths, especially, but also bnode syntax—make complex Stardog Rules concise and elegant.

NOTE

Starting in Stardog 3.0, it’s legal to use any valid Stardog built-in function in Stardog Rules.

How to Use Stardog Rules

There are three things to sort out:

1. Where to put these rules?
2. How to represent these rules?
3. What are the gotchas?

First, the rules go into the database, of course; and, in particular, they go into the named graph in which Stardog expects to find the TBox. This setting by default is the "default graph", i.e., unless you’ve changed the value of `reasoning.schema.graphs` , you’re probably going to be fine; that is, just add the rules to the database and it will all work out.[34]

Second, you represent the rules with specially constructed RDF triples. Here’s a kind of template example:

```
@prefix rule: <tag:stardog:api:rule:> .
[] a rule:SPARQLRule;
   rule:content """
   ...la di dah the rule goes here!
   """.
```

So there’s a namespace-- `tag:stardog:api:rule:` --that has a predicate, `content` , and a class, `SPARQLRule` . The object of this triple contains **one** rule in Stardog Rules syntax. A more realistic example:

```
@prefix rule: <tag:stardog:api:rule:> .

[] a rule:SPARQLRule ;
  rule:content """
    PREFIX :<urn:test:>
      IF {
            ?r a :Rectangle ;
```

```
                :width ?w ;
                :height ?h
            BIND (?w * ?h AS ?area)
          }
      THEN {
              ?r :area ?area
          }""" .
```

That’s pretty easy. Third, what are the gotchas?

1. The RDF serialization of rules in, say, a Turtle file has to use the `tag:stardog:api:rule:` namespace URI and then whatever prefix, if any, mechanism that’s valid for that serialization. In the examples here, we use Turtle. Hence, we use `@prefix`, etc.
2. However, the namespace URIs used **by the rules themselves** can be defined in only two places: the string that contains the rule—in the example above, you can see the default namespace is `urn:test:` --or in the Stardog database in which the rules are stored. Either place will work; if there are conflicts, the "closest definition wins", that is, if `foo:Example` is defined in both the rule content and in the Stardog database, the definition in the rule content is the one that Stardog will use.

Stardog Rules Examples

```
PREFIX rule: <tag:stardog:api:rule:>
PREFIX : <urn:test:>
PREFIX gr: <http://purl.org/goodrelations/v1#>

:Product1 gr:hasPriceSpecification [ gr:hasCurrencyValue 100.0 ] .
:Product2 gr:hasPriceSpecification [ gr:hasCurrencyValue 500.0 ] .
:Product3 gr:hasPriceSpecification [ gr:hasCurrencyValue 2000.0 ] .

[] a rule:SPARQLRule ;
   rule:content """
       PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
       PREFIX gr: <http://purl.org/goodrelations/v1#>
       PREFIX :<urn:test:>
     IF {
        ?offering gr:hasPriceSpecification ?ps .
        ?ps gr:hasCurrencyValue ?price .
        FILTER (?price >= 200.00).
     }
     THEN {
        ?offering a :ExpensiveProduct .
     }
   """.
```

This example is self-contained: it contains some data (the `:Product…` triples) and a rule. It also demonstrates the use of SPARQL's `FILTER` to do numerical (and other) comparisons.

Here's a more complex example that includes four rules and, again, some data.

```
PREFIX rule: <tag:stardog:api:rule:>
PREFIX : <urn:test:>

:c a :Circle ;
   :radius 10 .

:t a :Triangle ;
   :base 4 ;
   :height 10 .

:r a :Rectangle ;
   :width 5 ;
   :height 8 .

:s a :Rectangle ;
   :width 10 ;
   :height 10 .

[] a rule:SPARQLRule ;
   rule:content """
     PREFIX :<urn:test:>
     IF {
        ?r a :Rectangle ;
           :width ?w ;
           :height ?h
        BIND (?w * ?h AS ?area)
     }
     THEN {
         ?r :area ?area
     }""" .

[] a rule:SPARQLRule ;
```

```
   rule:content """
     PREFIX :<urn:test:>
     IF {
        ?t a :Triangle ;
           :base ?b ;
           :height ?h
        BIND (?b * ?h / 2 AS ?area)
     }
     THEN {
         ?t :area ?area
     }""" .

[] a rule:SPARQLRule ;
   rule:content """
     PREFIX :<urn:test:>
     PREFIX math: <http://www.w3.org/2005/xpath-functions/math#>
     IF {
          ?c a :Circle ;
             :radius ?r
          BIND (math:pi() * math:pow(?r, 2) AS ?area)
     }
     THEN {
         ?c :area ?area
     }""" .


[] a rule:SPARQLRule ;
   rule:content """
     PREFIX :<urn:test:>
     IF {
          ?r a :Rectangle ;
             :width ?w ;
             :height ?h
          FILTER (?w = ?h)
     }
     THEN {
         ?r a :Square
     }""" .
```

This example also demonstrates how to use SPARQL’s `BIND` to introduce intermediate variables and do calculations with or to them.

Let’s look at some other rules, but just the rule content this time for concision, to see some use of other SPARQL features.

This rule says that a person between 13 and 19 (inclusive) years of age is a teenager:

```
PREFIX swrlb: <http://www.w3.org/2003/11/swrlb#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

IF {
      ?x a :Person; hasAge ?age.
      FILTER (?age >= 13 && ?age <= 19)
}
THEN {
      ?x a :Teenager.
}
```

This rule says that a male person with a sibling who is the parent of a female is an "uncle with a niece":

```
IF {
      $x a Person; a :Male; :hasSibling $y;
      $y :isParentOf $z;
      $z a :Female.
}
THEN {
      $x a :UncleOfNiece.
}
```

We can use SPARQL 1.1 property paths (and bnodes for unnecessary variables (that is, ones that aren’t used in the `THEN`)) to render this rule even more concisely:

```
IF {
      $x a :Person, :Male; :hasSibling/:isParentOf [a :Female]
}
THEN {
      $x a :UncleOfNiece.
}
```

Aside: that’s pure awesome.

And of course a person who’s male and has a niece or nephew is an uncle of his niece(s) and nephew(s):

```
IF {
     ?x a :Male; :isSiblingOf/:isParentOf ?z
}
THEN {
      ?x :isUncleOf ?z.
}
```

Next rule example: a super user can read all of the things!

```
IF {
      ?x a :SuperUser.
      ?y a :Resource.
      ?z a <http://www.w3.org/ns/sparql#UUID>.
}
THEN {
      ?z a :Role.
      ?x :hasRole ?z; :readPermission ?y.
}
```

Supported Built-Ins

Stardog supports a wide variety of functions from SPARQL, XPath, SWRL, and some native Stardog functions, too. All of them may be used in either Stardog Rules syntax or in SWRL syntax. The supported functions are enumerated here.

Special Predicates

Stardog supports some builtin predicates with special meaning in order to make queries and rules easier to read and write. These special predicates are primarily syntactic sugar for more complex structures.

Direct/Strict Subclasses, Subproperties, & Direct Types

Besides the standard RDF(S) predicates `rdf:type` , `rdfs:subClassOf` and `rdfs:subPropertyOf` , Stardog supports the following special built-in predicates:

- `sp:directType`
- `sp:directSubClassOf`
- `sp:strictSubClassOf`
- `sp:directSubPropertyOf`
- `sp:strictSubPropertyOf`

Where the `sp` prefix binds to `tag:stardog:api:property:` . Stardog also recognizes `sesame:directType` , `sesame:directSubClassOf` , and `sesame:strictSubClassOf` predicates where the prefix `sesame` binds to `http://www.openrdf.org/schema/sesame#` .

We show what these each of these predicates means by relating them to an equivalent triple pattern; that is, you can just write the predicate rather than the (more unwieldy) triple pattern.

```
#c1 is a subclass of c2 but not equivalent to c2

:c1 sp:strictSubClassOf :c2      =>       :c1 rdfs:subClassOf :c2 .
                                          FILTER NOT EXISTS {
                                             :c1 owl:equivalentClass :c2 .
                                          }

#c1 is a strict subclass of c2 and there is no c3 between c1 and c2 in
#the strict subclass hierarchy

:c1 sp:directSubClassOf :c2      =>       :c1 sp:strictSubClassOf :c2 .
                                          FILTER NOT EXISTS {
                                             :c1 sp:strictSubClassOf :c3 .
                                             :c3 sp:strictSubClassOf :c2 .
                                          }

#ind is an instance of c1 but not an instance of any strict subclass of c1

:ind sp:directType :c1           =>       :ind rdf:type :c1 .
                                          FILTER NOT EXISTS {
                                             :ind rdf:type :c2 .
                                             :c2 sp:strictSubClassOf :c1 .
                                          }
```

The predicates `sp:directSubPropertyOf` and `sp:strictSubPropertyOf` are defined analogously.

New Individuals with SWRL

Stardog also supports a special predicate that extends the expressivity of SWRL rules. According to SWRL, you can’t create new individuals (i.e., new instances of classes) in a SWRL rule.

> **NOTE** Don’t get hung up by the tech vocabulary here..."new individual" just means that you can’t have a rule that creates a new instance of some RDF or OWL class as a result of the rule firing.

This restriction is well-motivated; without it, you can easily create rules that do not terminate, that is, never reach a fixed point. Stardog’s user-defined rules weakens this restriction in some crucial aspects, subject to the following restrictions, conditions, and warnings.

> **WARNING** This special predicate is basically a loaded gun with which you may shoot yourselves in the foot if you aren’t very careful.

So despite the general restriction in SWRL, in Stardog we actually can create new individuals with a rule by using the function `UUID()` as follows:

```
IF {
    ?p a :Parent .
    BIND (UUID() AS ?parent) .
}
THEN {
    ?parent a :Person .
}
```

NOTE Alternatively, we can use the predicate `http://www.w3.org/ns/sparql#UUID` as a unary SWRL built-in.

This rule will create a **random** URI for each instance of the class `:Parent` and also assert that each new instance is an instance of `:Person` --parents are people, too!

Remarks

1. The URIs for the generated individuals are meaningless in the sense that they should not be used in further queries; that is to say, these URIs are not guaranteed by Stardog to be stable.
2. Due to normalization, rules with more than one atom in the head are broken up into several rules.

Thus,

```
IF {
    ?person a :Person .
    BIND (UUID() AS ?parent) .
}
THEN {
    ?parent a :Parent ;
            a :Male .
}
```

will be normalized into two rules:

```
IF {
    ?person a :Person .
    BIND (UUID() AS ?parent) .
}
THEN {
    ?parent a :Parent .
}

IF {
    ?person a :Person .
    BIND (UUID() AS ?parent) .
}
```

```
THEN {
    ?parent a :Male .
}
```

As a consequence, instead of stating that the new individual is both an instance of `:Male` and `:Parent`, we would create two **different** new individuals and assert that one is male and the other is a parent. If you need to assert various things about the new individual, we recommend the use of extra rules or axioms. In the previous example, we can introduce a new class (`:Father`) and add the following rule to our schema:

```
IF {
    ?person a :Father .
}
THEN {
    ?parent a :Parent ;
            a :Male .
}
```

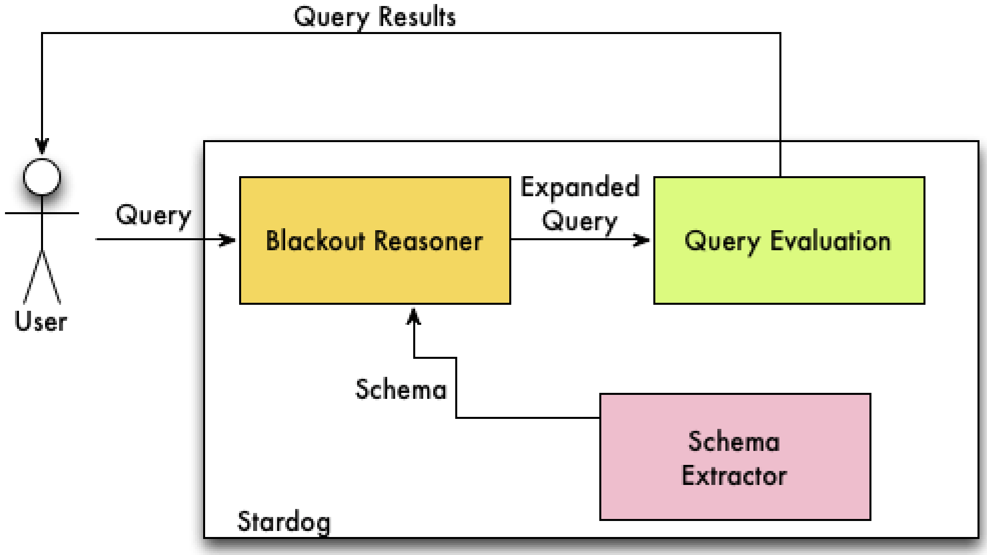
And then modify the original rule accordingly:

```
IF {
    ?person a :Person .
    BIND (UUID() AS ?parent) .
}
THEN {
    ?parent a :Father .
}
```

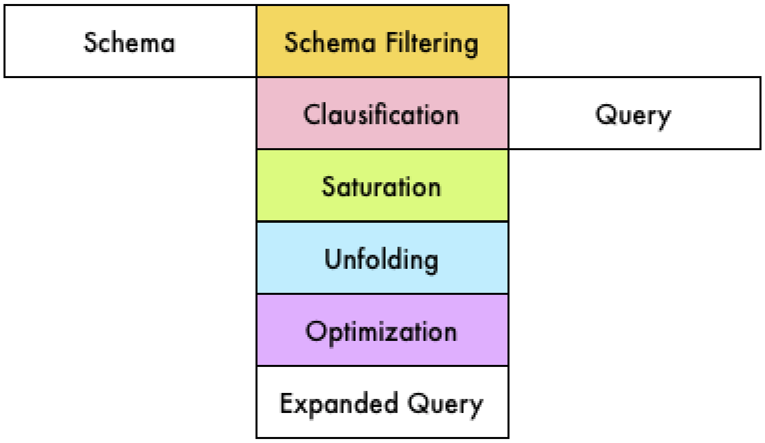
Query Rewriting

Reasoning in Stardog is based (mostly) on a **query rewriting** technique: Stardog rewrites the user’s query with respect to any schema or rules, and then executes the resulting expanded query (EQ) against the data in the normal way. This process is completely automated and requires no intervention from the user.

As can be seen in Figure 1, the rewriting process involves five different phases.



1. Figure 1 Query Answering



2. Figure 2. Query Rewriting

We illustrate the query answering process by means of an example. Consider a Stardog database, MyDB$_1$, containing the following schema:

```
:SeniorManager rdfs:subClassOf :manages some :Manager
:manages some :Employee rdfs:subClassOf :Manager
:Manager rdfs:subClassOf :Employee
```

Which says that a senior manager manages at least one manager, that every person that manages an employee is a manager, and that every manager is also an employee.

Let’s also assume that MyDB$_1$ contains the following data assertions:

```
:Bill rdf:type :SeniorManager
:Robert rdf:type :Manager
:Ana :manages :Lucy
:Lucy rdf:type :Employee
```

Finally, let’s say that we want to retrieve the set of all employees. We do this by posing the following query:

```
SELECT ?employee WHERE { ?employee rdf:type :Employee }
```

To answer this query, Stardog first **rewrites** it using the information in the schema. So the original query is rewritten into four queries:

```
SELECT ?employee WHERE { ?employee rdf:type :Employee }
SELECT ?employee WHERE { ?employee rdf:type :Manager }
SELECT ?employee WHERE { ?employee rdf:type :SeniorManager }
SELECT ?employee WHERE { ?employee :manages ?x. ?x rdf:type :Employee }
```

Then Stardog executes these queries over the data as if they were written that way to begin with. In fact, Stardog can’t tell that they weren’t. Reasoning in Stardog **just is query answering** in nearly every case.

The form of the EQ depends on the reasoning type. For OWL 2 QL, every EQ produced by Stardog is **guaranteed to be expanded into a set of queries**. If the reasoning type is OWL 2 RL or EL, then the EQ **may** (but may not) include a recursive rule. *If a recursive rule is included, Stardog’s answers may be incomplete with respect to the semantics of the reasoning type.*

Why Query Rewriting?

Query rewriting has several advantages over materialization. In materialization, the data gets expanded with respect to the schema, not with respect to any actual query. And it’s the data—all of the data—that gets expanded, whether any actual query subsequently requires reasoning or not. The schema is used to generate new triples, typicaly when data is added or removed from the system. However, materialization introduces several thorny issues:

1. **data freshness**. Materialization has to be performed **every time** the data or the schema change. This is particularly unsuitable for applications where the data changes frequently.
2. **data size**. Depending on the schema, materialization can significantly increase the size of the data, sometimes dramatically so. The cost of this data size blowup may be applied to **every** query in terms of increased I/O.
3. **OWL 2 profile reasoning**. Given the fact that QL, RL, and EL are not comparable with respect to expressive power, an application that requires reasoning with more than one profile would need to maintain different corresponding materialized versions of the data.
4. **Resources**. Depending on the size of the original data and the complexity of the schema, materialization may be computationally expensive. And truth maintenance, which materialization requires, is **always** computationally expensive.

Same As Reasoning

Stardog 3.0 adds full support for OWL 2 `sameAs` reasoning. However, `sameAs` reasoning works in a different way than the rest of the reasoning mechanism. The `sameAs` inferences are computed and indexed eagerly so that these materialized inferences can be used directly at query rewriting time. The `sameAs` index is updated automatically as the database is modified so the difference is not of much direct concern to users.

In order to use `sameAs` reasoning, the database configuration option `reasoning.sameas` should be set either at database creation time or at a later time when the database is offline. This can be done through the Web Console or using the command line as follows:

```
$ ./stardog-admin db create -o reasoning.sameas=FULL -n myDB
```

There are legal three values for this option:

- `OFF` disables all `sameAs` inferences, that is, only asserted `sameAs` triples will be included in query results.[35]
- `ON` computes `sameAs` inferences using only asserted `sameAs` triples, considering the reflexivity, symmetry and transitivity of the `sameAs` relation.
- `FULL` same as `ON` but also considers OWL functional properties, inverse functional properties, and `hasKey` axioms while computing `sameAs` inferences.

> **NOTE**
>
> The way `sameAs` reasoning works differs from the OWL semantics slightly in the sense that Stardog designates one canonical individual for each `sameAs` equivalence set and only returns the canonical individual. This avoids the combinatorial explosion in query results while providing the data integration benefits.

Let’s see an example showing how `sameAs` reasoning works. Consider the following database where `sameAs` reasoning is set to `ON`:

```
dbpedia:Elvis_Presley
    dbpedia-owl:birthPlace dbpedia:Mississippi ;
    owl:sameAs freebase:en.elvis_presley .

nyt:presley_elvis_per
    nyt:associated_article_count 35 ;
    rdfs:label "Elvis Presley" ;
    owl:sameAs dbpedia:Elvis_Presley .

freebase:en.elvis_presley
        freebase:common.topic.official_website <http://www.elvis.com/> .
```

Now consider the following query and its results:

```
$ ./stardog query --reasoning elvis 'SELECT * { ?s dbpedia-owl:birthPlace ?o;
rdfs:label "Elvis Presley" }'
```

```
+-----------------------+---------------------+
|           s           |          o          |
+-----------------------+---------------------+
| nyt:presley_elvis_per | dbpedia:Mississippi |
+-----------------------+---------------------+
```

Let’s unpack this carefully. There are three things to note.

First, the query returns only one result even though there are three different URIs that denote Elvis Presley. Second, the URI returned is fixed but chosen randomly. **Stardog picks one of the URIs as the canonical URI and always returns that and only that canonical URI in the results.** If more `sameAs` triples are added the chosen canonical individual may change. Third, it is important to point out that even though only one URI is returned, the effect of `sameAs` reasoning is visible in the results since the `rdfs:label` and `dbpedia-owl:birthPlace` properties were asserted about different instances (i.e., different URIs).

Now, you might might be inclined to write queries such as this to get all the properties for a specific URI:

```
SELECT * {
   nyt:presley_elvis_per owl:sameAs ?elvis .
   ?elvis ?p ?o
}
```

However, this is completely unnecessary; rather, you can write the following query and get the same results since `sameAs` reasoning would automatically merge the results for you. Therefore, the query

```
SELECT * {
   nyt:presley_elvis_per ?p ?o
}
```

would return these results:

```
+----------------------------------------+-----------------------+
|                   p                    |           o           |
+----------------------------------------+-----------------------+
| rdfs:label                             | "Elvis Presley"       |
| dbpedia-owl:birthPlace                 | dbpedia:Mississippi   |
| nyt:associated_article_count           | 35                    |
| freebase:common.topic.official_website | http://www.elvis.com/ |
| rdf:type                               | owl:Thing             |
+----------------------------------------+-----------------------+
```

NOTE The URI used in the query does not need to be the same one returned in the results. Thus, the following query would return the exact same results, too:

```
SELECT * {
   dbpedia:Elvis_Presley nyt:presley_elvis_per ?p ?o
}
```

The only time Stardog will return a non-canonical URI in the query results is when you explicitly query for the `sameAs` inferences as in this next example:

```
$ ./stardog query -r elvis 'SELECT * { freebase:en.elvis_presley owl:sameAs ?elvis }'
```

```
+---------------------------+
|           elvis           |
+---------------------------+
| dbpedia:Elvis_Presley     |
| freebase:en.elvis_presley |
| nyt:presley_elvis_per     |
+---------------------------+
```

In the `FULL` `sameAs` reasoning mode, Stardog will also take other OWL axioms into account when computing `sameAs` inferences. Consider the following example:

```
#Everyone has a unique SSN number
:hasSSN a owl:InverseFunctionalProperty , owl:DatatypeProperty .
```

```
:JohnDoe :hasSSN "123-45-6789" .
:JDoe :hasSSN "123-45-6789" .

#Nobody can work for more than one company (for the sake of the example)
:worksFor a owl:FunctionalProperty , owl:ObjectProperty ;
        rdfs:domain :Employee ;
        rdfs:range :Company .

:JohnDoe :worksFor :Acme .
:JDoe :worksFor :AcmeInc .

#For each company, there can only be one employee with the same employee ID
:Employee owl:hasKey (:employeeID :worksFor ).

:JohnDoe :employeeID "1234-ABC" .

:JohnD :employeeID "1234-ABC" ;
       :worksFor :AcmeInc .

:JD :employeeID "5678-XYZ" ;
    :worksFor :AcmeInc .

:John :employeeID "1234-ABC" ;
      :worksFor :Emca .
```

For this database, with `sameAs` reasoning set to `FULL`, we would get the following answers:

```
$ ./stardog query -r acme "SELECT * {?x owl:sameAs ?y}"
```

```
+----------+----------+
|    x     |    y     |
+----------+----------+
| :JohnDoe | :JohnD   |
| :JDoe    | :JohnD   |
| :Acme    | :AcmeInc |
+----------+----------+
```

We can follow the chain of inferences to understand how these results were computed:

1. `:JohnDoe owl:sameAs :JohnD` can be computed due to the fact that both have the same SSN numbers and `hasSSN` property is inverse functional.
2. We can infer `:Acme owl:sameAs :AcmeInc` since `:JohnDoe` can work for at most one company.
3. `:JohnDoe owl:sameAs :JohnD` can be inferred using the `owl:hasKey` definition since both individuals are known to work for the same company and have the same employee ID.
4. No more `sameAs` inferences can be computed due to the key definition, since other employees either have different IDs or work for other companies.

Removing Unwanted Inferences

Sometimes reasoning can produce unintended inferences. Perhaps there are modeling errors in the schema or incorrect assertions in the data. After an unintended inference is detected, it might be hard to figure out how to fix it, because there might be multiple different reasons for the inference. The `reasoning explain` command can be used to see the different explanations and the `reasoning undo` command can be used to generate a SPARQL update query that will remove the minimum amount of triples necessary to remove the unwanted inference:

```
$ ./reasoning undo myDB ":AcmeInc a :Person"
```

Performance Hints

The query rewriting approach suggests some guidelines for more efficient query answering.

Hierarchies and Queries

Avoid unnecessarily deep class/property hierarchies.

> If you do not need to model several different types of a given class or property in your schema, then don’t do that! The reason shallow hierarchies are desirable is that the maximal hierarchy depth in the schema partly determines the maximal size of the EQs produced by Stardog. The larger the EQ, the longer it takes to evaluate, generally.

For example, suppose our schema contains a very thorough and detailed set of subclasses of the class `:Employee`:

```
:Manager rdfs:subClassOf :Employee
:SeniorManager rdfs:subClassOf :Manager
...

:Supervisor rdfs:subClassOf :Employee
:DepartmentSupervisor rdfs:subClassOf :Supervisor
...

:Secretary rdfs:subClassOf :Employee
...
```

If we wanted to retrieve the set of all employees, Stardog would produce an EQ containing a query of the following form for every subclass `:Ci` of `:Employee`:

```
SELECT ?employee WHERE { ?employee rdf:type :Ci }
```

Thus, **ask the most specific query sufficient for your use case**. Why? More general queries—that is, queries that contain concepts high up in the class hierarchy defined by the schema—will typically yield larger EQs.

Domains and Ranges

Specify domain and range of the properties in the schema.

These types of axiom can improve query performance significantly. Consider the following query asking for people and the employees they manage:

```
SELECT ?manager ?employee WHERE
  { ?manager :manages ?employee.
    ?employee rdf:type :Employee. }
```

We know that this query would cause a large EQ given a deep hierarchy of `:Employee` subclasses. However, if we added the following single range axiom:

```
:manages rdfs:range :Employee
```

then the EQ would collapse to

```
 SELECT ?manager ?employee WHERE { ?manager :manages ?employee }
```

which is considerably easier to evaluate.

Very Large Schemas

If you are working with a very large schema like SNOMED then there are couple things to note. First of all, Stardog reasoning works by pulling the complete schema into memory. This means you might need to increase the default memory settings for Stardog for a large schema. Stardog performs all schema reasoning upfront and only once but waits until the first reasoning query arrives. With a large schema, this step can be slow but subsequent reasoning queries will be fast. Also note that, Stardog will update schema reasoning results automatically after the database is modified so there will be some processing time spent then.

Reasoning with very expressive schemas can be time consuming and use a lot of memory. To get the best performance out of Stardog with large schemas, limit the expressivity of your schema to OWL 2 EL. You can also set the reasoning type of the database to `EL` and Stardog will automatically filter any axiom outside the EL expressivity. See Reasoning Types for more details on reasoning types. OWL 2 EL allows range declarations for properties and user-defined datatypes but avoiding these two constructs will further improve schema reasoning performance in Stardog.

Not Seeing Expected Results?

Here’s a few things that you might want to consider.

Are variable types ambiguous?

When a SPARQL query gets executed, each variable is bound to a URI, blank node, or to a literal to form a particular result (a collection of these results is a result set). In the context of reasoning, URIs might represent different entities: individuals, classes, properties, etc. According to the relevant standard, **every variable in a SPARQL query must bind to at most one of these types of entity**.

Stardog can often figure out the right entity type from the query itself (e.g., given the triple pattern `?i ?p "a literal"`, we know `?p` is supposed to bind to a data property); however, sometimes this isn't possible (e.g., `?s ?p ?o`). In case the types can't be determined automatically, **Stardog logs a message and evaluates the query by making some assumptions, which may not be what the query writer intended, about the types of variables**.

You can add one or more type triples to the query to resolve these ambiguities.[36]

These "type triples" have the form `?var a TYPE`, where `TYPE` is a URI representing the type of entity to which the variable `?var` is supposed to bind: the most common are `owl:ObjectProperty` or `owl:DatatypeProperty`; in some cases, you might want `owl:NamedIndividual`, or `owl:Class`. For instance, you can use the following query to retrieve all object properties and their characteristics; without the type triple, `?s` will bind only to individuals:

```
SELECT ?o
WHERE {
    ?s rdf:type ?o.
    ?s a owl:ObjectProperty.
}.
```

Since Stardog now knows that `?s` should bind to an object property, it can now infer that `?o` binds to property characteristics of `?s`.

Is the schema where you think it is?

Starting in Stardog 3.0, Stardog will extract the schema from **all named graphs and the default graph**.

If you require that the schema only be extracted from one or more specific named graphs, then you must tell Stardog where to find the schema. See database configuration options for details.

You can also use the `reasoning schema` command to export the contents of the schema to see exactly what is included in the schema that Stardog uses.

Are you using the right reasoning type?

Perhaps some of the modeling constructs (a.k.a. axioms) in your database are being ignored. By default, Stardog uses the `SL` reasoning type. You can find out which axioms are being ignored by looking at the Stardog log file.

Are you using DL?

Stardog supports full OWL 2 DL reasoning but only for data that fits into main memory.

Are you using SWRL?

SWRL rules—whether using SWRL syntax or Stardog Rules Syntax—are only taken into account using the **SL** reasoning type.

Do you know what to expect?

The OWL 2 primer is a good place to start.

Known Issues

Stardog 3.1.4 does not

- Follow ontology `owl:imports` statements automatically; any imported OWL ontologies that are required must be loaded into a Stardog database in the normal way.
- Handle recursive queries. If recursion is necessary to answer the query with respect to the schema, results will be sound (*no wrong answers*) but potentially incomplete (*some correct answers not returned*) with respect to the requested reasoning type.

Terminology

This chapter uses the following terms of art.

Databases

A **database** (DB), a.k.a. ontology, is composed of two different parts: the schema or **Terminological Box** (TBox) and the data or **Assertional Box** (ABox). Analogus to relational databases, the TBox can be thought of as the schema, and the ABox as the data. In other words, the TBox is a set of **axioms**, whereas the ABox is a set of **assertions**.

As we explain in OWL 2 Profiles, the kinds of assertion and axiom that one might use for a particular database are determined by the fragment of OWL 2 to which you'd like to adhere. In general, you should choose the OWL 2 profile that most closely fits the data modeling needs of your application.

The most common data assertions are class and property assertions. Class assertions are used to state that a particular individual is an instance of a given class. Property assertions are used to state that two particular individuals (or an individual and a literal) are related via a given property. For example, suppose we have a DB MyDB$_2$ that contains the following data assertions. We use the usual standard prefixes for RDF(S) and OWL.

```
:complexible rdf:type :Company
:complexible :maintains :Stardog
```

Which says that `:complexible` is a company, and that `:complexible` maintains `:Stardog`.

The most common schema axioms are subclass axioms. Subclass axioms are used to state that every instance of a particular class is also an instance of another class. For example, suppose that MyDB$_2$ contains the following TBox axiom:

```
:Company rdfs:subClassOf :Organization
```

stating that companies are a type of organization.

Queries

When reasoning is enabled, Stardog executes SPARQL queries depending on the type of Basic Graph Patterns they contain. A BGP is said to be an "ABox BGP" if it is of one of the following forms:

- **term$_1$** `rdf:type` **uri**
- **term$_1$ uri term$_2$**
- **term$_1$** `owl:differentFrom` **term$_2$**
- **term$_1$** `owl:sameAs` **term$_2$**

A BGP is said to be a TBox BGP if it is of one of the following forms:

- **term$_1$** `rdfs:subClassOf` **term$_2$**
- **term$_1$** `owl:disjointWith` **term$_2$**
- **term$_1$** `owl:equivalentClass` **term$_2$**
- **term$_1$** `rdfs:subPropertyOf` **term$_2$**
- **term$_1$** `owl:equivalentProperty` **term$_2$**
- **term$_1$** `owl:inverseOf` **term$_2$**
- **term$_1$** `owl:propertyDisjointWith` **term$_2$**
- **term$_1$** `rdfs:domain` **term$_2$**
- **term$_1$** `rdfs:range` **term$_2$**

A BGP is said to be a Hybrid BGP if it is of one of the following forms:

- **term$_1$** `rdf:type` **?var**
- **term$_1$?var term$_2$**

where **term** (possibly with subscripts) is either an URI or variable; **uri** is a URI; and **?var** is a variable.

When executing a query, ABox BGPs are handled by Stardog. TBox BGPs are executed by Pellet embedded in Stardog. Hybrid BGPs by a combination of both.

Reasoning

Intuitively, reasoning with a DB means to make implicit knowledge explicit. There are two main use cases for reasoning: to infer implicit knowledge and to discover modeling errors.

With respect to the first use case, recall that MyDB$_2$ contains the following assertion and axiom:

```
 :complexible rdf:type :Company
 :Company rdfs:subClassOf :Organization
```

From this DB, we can use Stardog in order to **infer** that `:complexible` is an organization:

```
:complexible rdf:type :Organization
```

Using reasoning in order to infer implicit knowledge in the context of an enterprise application can lead to simpler queries. Let us suppose, for example, that MyDB$_2$ contains a complex class hierarchy including several types of organization (including company). Let us further suppose that our application requires to use Stardog in order to get the list of all considered organizations. If Stardog were used **with reasoning**, then we would need only issue the following simple query:

```
SELECT ?org WHERE { ?org rdf:type :Organization}
```

In contrast, if we were using Stardog **with no reasoning**, then we would have to issue a more complex query that considers all possible types of organization, thus coupling queries to domain knowledge in a tight way:

```
SELECT ?org WHERE
              { { ?org rdf:type :Organization } UNION
              { ?org rdf:type :Company } UNION
...
}
```

Which of these queries seems more loosely coupled and more resilient to change?

Stardog can also be used in order to discover modeling errors in a DB. The most common modeling errors are **unsatisfiable** classes and **inconsistent** DBs.

An unsatisfiable class is simply a class that cannot have any instances. Say, for example, that we added the following axioms to MyDB$_2$:

```
:Company owl:disjointWith :Organization
:LLC owl:equivalentClass :Company and :Organization
```

stating that companies cannot be organizations and vice versa, and that an LLC is a company and an organization. The disjointness axiom causes the class `:LLC` to be unsatisfiable because, for the DB to be free of any logical contradiction, there can be no instances of `:LLC`.

Asserting (or inferring) that an unsatisfiable class has an instance, causes the DB to be **inconsistent**. In the particular case of MyDB$_2$, we know that `:complexible` is a company **and** an organization; therefore, we also know that it is an instance of `:LLC`, and as `:LLC` is known to be unsatisfiable, we have that MyDB$_2$ is inconsistent.

Using reasoning in order to discover modeling errors in the context of an enterprise application is useful in order to maintain a correct contradiction-free model of the domain. In our example, we discovered that `:LLC` is unsatisfiable and MyDB$_2$ is inconsistent, which leads us to believe that there is a modeling error in our DB. In this case, it is easy to see that the problem is the disjointness axiom between `:Company` and `:Organization`.

OWL 2 Profiles

As explained in the OWL 2 Web Ontology Language Profiles Specification, an OWL 2 profile is a reduced version of OWL 2 that trades some expressive power for efficiency of reasoning. There are three OWL 2 profiles, each of which achieves efficiency differently.

- OWL 2 QL is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. The expressive power of the profile is necessarily limited; however, it includes most of the main features of conceptual models such as UML class diagrams and ER diagrams.

- OWL 2 EL is particularly useful in applications employing ontologies that contain very large numbers of properties and classes. This profile captures the expressive power used by many such ontologies and is a subset of OWL 2 for which the basic reasoning problems can be performed in time that is polynomial with respect to the size of the ontology.
- OWL 2 RL is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate OWL 2 applications that can trade the full expressivity of the language for efficiency, as well as RDF(S) applications that need some added expressivity.

Each profile restricts the kinds of axiom and assertion that can be used in a DB. Colloquially, QL is the least expressive of the profiles, followed by RL and EL; however, strictly speaking, no profile is more expressive than any other as they provide incomparable sets of constructs.

Stardog supports the three profiles of OWL 2. Notably, since TBox BGPs are handled completely by Pellet, Stardog supports reasoning for the whole of OWL 2 for queries containing TBox BGPs only.

Validating Constraints

Stardog Integrity Constraint Validation ("ICV") validates RDF data stored in a Stardog database according to data rules (i.e., "constraints") described by users and that make sense for their domain, application, and data. These constraints may be written in SPARQL, OWL, or SWRL. This chapter explains how to use ICV.

The use of high-level languages (OWL 2, SWRL, and SPARQL) to validate RDF data using **closed world semantics** is one of Stardog’s unique capabilities. Using high level languages like OWL, SWRL, and SPARQL as schema or constraint languages for RDF and Linked Data has several advantages:

- Unifying the domain model with data quality rules

- Aligning the domain model and data quality rules with the integration model and language (i.e., RDF)
- Being able to query the domain model, data quality rules, integration model, mapping rules, etc with SPARQL
- Being able to use automated reasoning about all of these things to insure logical consistency, explain errors and problems, etc.

If you are also interested in theory and background, please see the ICV specification, which has all the formal details.

Getting Started with ICV

This log of a CLI session gives a full example of how to validate data using a mix of integrity constraints expressed in OWL and SPARQL. It uses data and constraints linked below.

```
# Stardog commands and the output for RDF validation example

# First create the Stardog database and load data

$ ./stardog-admin db create -n sota sota-data.ttl
Bulk loading data to new database.
Loading data completed...Loaded 23 triples in 00:00:00 @ 0.4K triples/sec.
Successfully created database 'sota'.

# Then add the constraints to the database

$ ./stardog-admin icv add sota sota-constraints.ttl
Successfully added constraints in 00:00:00.

# Now run the validation command
# This command just prints which constraints are violated, see the Java
# example for printing the details about validation

$ ./stardog icv validate sota
Data is NOT valid.
The following constraints were violated:
AxiomConstraint{:related rdfs:range :Issue}
AxiomConstraint{:reportedOn rdfs:domain :Issue}
AxiomConstraint{:Issue rdfs:subClassOf (:reportedBy exactly 1 owl:Thing)}
```

```
AxiomConstraint{:Issue rdfs:subClassOf (:reproducedBy min 0 owl:Thing)}
AxiomConstraint{:reproducedBy rdfs:range foaf:Person}
AxiomConstraint{:reportedBy rdfs:range foaf:Person}
AxiomConstraint{:Issue rdfs:subClassOf (:related min 0 owl:Thing)}
AxiomConstraint{:state rdfs:domain :Issue}
AxiomConstraint{:state rdfs:range :ValidState}
AxiomConstraint{:Issue rdfs:subClassOf (:reproducedOn min 0 rdfs:Literal)}


# We can also add SPARQL queries as constraints

$ ./stardog-admin icv add sota-query.sparql

# We can run validation with a mixture of OWL constraints and SPARQL constraints

$ ./stardog icv validate sota
Data is NOT valid.
...
```

See the following Gists to follow along at home:

- Constraints in OWL
- Example Data in Turtle
- Java Example
- Example Output
- SPARQL Query Example in Java
- SPARQL Query Output
- Constraints in SPARQL

And, finally, the full Gist with links to everything. In the rest of this chapter, we explain in more detail about programmatic access, as well as give a full slate of examples of ICV in action.

ICV & OWL 2 Reasoning

An integrity constraint may be satisfied or violated in either of two ways: by an explicit statement in a Stardog database or by a statement that's been validly inferred by Stardog.

When ICV is enabled for a Stardog database, it has to be enabled with respect to a reasoning type or level. The valid choices of reasoning type are any type or kind of reasoning supported by Stardog. See OWL & Rule Reasoning for the details.

So ICV is performed with three inputs:

1. a Stardog database,
2. a set of constraints, and
3. a reasoning type (which may be, of course, no reasoning).

This is the case because domain modelers, ontology developers, or integrity constraint authors must consider the interactions between explicit and inferred statements and how these are accounted for in integrity constraints.

Using ICV from CLI

To add constraints to a database:

```
$ stardog-admin icv add myDb constraints.rdf
```

To drop all constraints from a database:

```
$ stardog-admin icv drop myDb
```

To remove one or more specific constraints from a database:

```
$ stardog-admin icv remove myDb constraints.rdf
```

To convert new or existing constraints into SPARQL queries for export:

```
$ stardog icv convert myDb out.rdf
```

To explain a constraint violation:

```
$ stardog explain --contexts http://example.org/context1 http://example.org/context2
```

To export constraints:

```
$ stardog icv export myDb constraints.rdf
```

To validate a database (or some named graphs) with respect to constraints:

```
$ stardog validate --contexts http://example.org/context1 http://example.org/context2
```

ICV Guard Mode

Stardog will also apply constraints as part of its transactional cycle and fail transactions that violate constraints. We call this "guard mode". It must be enabled explicitly in the database configuration options. Using the command line, these steps are as follows:

```
$ ./stardog-admin db offline --timeout 0 myDb #take the database offline
$ ./stardog-admin db metadata set icv.enabled=true myDb #enable ICV
$ ./stardog-admin db online myDb #put the database online
```

In the Web Console you can set the database offline, click [**Edit**], change the "ICV Enable" value, click [**Save**] and set the database online again.

Once guard mode is enabled, modifications of the database (via SPARQL Update or any other method), whether adds or deletes, that violate the integrity constraints will cause the transaction to fail.

Explaining ICV Violations

ICV violations can be explained using Stardog's Proof Trees. The following command will explain the IC violations for constraints stored in the database:

```
$ stardog icv explain --reasoning "myDB"
```

It is possible to explain violations for external constraints by passing the file with constraints as an additional argument:

```
$ stardog icv explain --reasoning "myDB" constraints.ttl
```

Security Note

WARNING

> There is a security implication in this design that may not be obvious. Changing the reasoning type associated with a database and integrity constraint validation may have serious security implications with respect to a Stardog database and, thus, may only be performed by a user role with sufficient privileges for that action.

Repairing ICV Violations

Stardog 3.0 adds support for automatic repair of some kinds of integrity violation. This can be accomplished programmatically via API, as well as via CLI using the `icv fix` subcommand.

```
$ stardog help icv fix
```

Repair plans are emitted as a sequence of SPARQL Update queries, which means they can be applied to any system that understands SPARQL Update. If you pass `--execute` the repair plan will be applied immediately.

`icv fix` will repair violations of all constraints in the database; if you'd prefer to fix the violations for only some constraints, you can pass those constraints as an additional argument. Although a possible (but trivial) fix for any violation is to remove one or more constraints, `icv fix` does not suggest that kind of repair, even though it may be appropriate in some cases.

ICV Examples

Stardog ICV has a formal semantics. But let's just look at some examples instead; these examples use OWL 2 Manchester syntax, and they assume a simple data schema, which is available as an OWL ontology and as a UML diagram. The examples assume that the default namespace is `http://example.com/company.owl#` and that `xsd:` is bound to the standard, `http://www.w3.org/2001/XMLSchema#`.

Reference Java code is available for each of the following examples and is also distributed with Stardog.

Subsumption Constraints

This kind of constraint guarantees certain subclass and superclass (i.e., subsumption) relationships exist between instances.

Managers must be employees.

Constraint

```
:Manager rdfs:subClassOf :Employee
```

Database A (invalid)

```
:Alice a :Manager .
```

Database B (valid)

```
:Alice a :Manager , :Employee .
```

This constraint says that if an RDF individual is an instance of `Manager`, then it must also be an instance of `Employee`. In A, the only instance of `Manager`, namely `Alice`, is not an instance of `Employee`; therefore, A is invalid. In B, `Alice` is an instance of Database both `Manager` and `Employee`; therefore, B is valid.

Domain-Range Constraints

These constraints control the types of domain and range instances of properties.

Only project leaders can be responsible for projects.

Constraint

```
:is_responsible_for rdfs:domain :Project_Leader
```

Database A (invalid)

```
:Alice :is_responsible_for :MyProject .

:MyProject a :Project .
```

Database B (invalid)

```
:Alice a :Project_Leader ;
        :is_responsible_for :MyProject .
```

Database C (valid)

```
:Alice a :Project_Leader ;
        :is_responsible_for :MyProject .

:MyProject a :Project .
```

This constraint says that if two RDF instances are related to each other via the property `is_responsible_for`, then the range instance must be an instance of `Project_Leader` and the domain instance must be an instance of `Project`. In Database A, there is only one pair of individuals related via `is_responsible_for`, namely `(Alice, MyProject)`, and `MyProject` is an instance of `Project`; but `Alice` is not an instance of `Project_Leader`. Therefore, A is invalid. In B, `Alice` is an instance of `Project_Leader`, but `MyProject` is not an instance of `Project`; therefore, B is not valid. In C, `Alice` is an instance of `Project_Leader`, and `MyProject` is an instance of `Project`; therefore, C is valid.

Only employees can have an SSN.

Constraint

```
:ssn rdfs:domain :Employee
```

Database A (invalid)

```
:Bob :ssn "123-45-6789" .
```

Database B (valid)

```
:Bob a :Employee ;
        :ssn "123-45-6789" .
```

This constraint says that if an RDF instance `i` has a data assertion via the the property `SSN`, then `i` must be an instance of `Employee`. In A, `Bob` is not an instance of `Employee` but has `SSN`; therefore, A is invalid. In B, `Bob` is an instance of `Employee`; therefore, B is valid.

A date of birth must be a date.

Constraint

```
:dob rdfs:range xsd:date
```

Database A (invalid)

```
:Bob :dob "1970-01-01" .
```

Database B (valid)

```
:Bob :dob "1970-01-01"^^xsd:date
```

This constraint says that if an RDF instance `i` is related to a literal `l` via the data property `DOB`, then `l` must have the XML Schema type `xsd:date`. In A, `Bob` is related to the untyped literal `"1970-01-01"` via `DOB` so A is invalid. In B, the literal `"1970-01-01"` is properly typed so it's valid.

Participation Constraints

These constraints control whether or not an RDF instance participates in some specified relationship.

Each supervisor must supervise at least one employee.

Constraint

```
#this constraint is very concise in Terp syntax:
#:Supervisor rdfs:subClassOf (:supervises some :Employee)

:Supervisor rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty :supervises ;
                owl:someValuesFrom :Employee
              ] .
```

Database A (valid)

```
:Alice a owl:Thing .
```

Database B (invalid)

```
:Alice a :Supervisor .
```

Database C (invalid)

```
:Alice a :Supervisor ;
        :supervises :Bob .
```

Database D (valid)

```
:Alice a :Supervisor ;
        :supervises :Bob .

:Bob a :Employee
```

This constraint says that if an RDF instance `i` is of type `Supervisor`, then `i` must be related to an individual `j` via the property `supervises` and also that `j` must be an instance of `Employee`. In A, `Supervisor` has no instances; therefore, A is trivially valid. In B, the only instance of `Supervisor`, namely `Alice`, is related to no individual; therefore, B is invalid. In C, `Alice` is related to `Bob` via `supervises`, but `Bob` is not an instance of `Employee`; therefore, C is invalid. In D, `Alice` is related to `Bob` via `supervises`, and `Bob` is an instance of `Employee`; hence, D is valid.

Each project must have a valid project number.

Constraint

```
#Again, this constraint in Terp syntax rocks the hizzous:
#:Project rdfs:subClassOf (:number some xsd:integer[>= 0, < 5000])

:Project rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty :number ;
                owl:someValuesFrom
                        [ a rdfs:Datatype ;
                          owl:onDatatype xsd:integer ;
                          owl:withRestrictions ([xsd:minInclusive 0] [
xsd:maxExclusive 5000])
                        ]
              ] .
```

Database A (valid)

```
:MyProject a owl:Thing .
```

Database B (invalid)

```
:MyProject a :Project
```

Database C (invalid)

```
:MyProject a :Project ;
        :number "23" .
```

Database D (invalid)

```
:MyProject a :Project ;
        :number "6000"^^xsd:integer .
```

Database E (valid)

```
:MyProject a :Project ;
        :number "23"^^xsd:integer .
```

This constraint says that if an RDF instance `i` is of type `Project`, then `i` must be related via the property `number` to an integer between `0` and `5000` (inclusive)—that is, projects have project numbers in a certain range. In A, the individual `MyProject` is not known to be an instance of `Project` so the constraint does not apply at all and A is valid. In B, `MyProject` is an instance of `Project` but doesn't have any data assertions via `number` so A is invalid. In C, `MyProject` does have a data property assertion via `number` but the literal `"23"` is untyped—that is, it's not an integer—therefore, C is invalid. In D, `MyProject` is related to an integer via `number` but it is out of the range: D is invalid. Finally, in E, `MyProject` is related to the integer `23` which is in the range of `[0,5000]` so E is valid.

Cardinality Constraints

These constraints control the number of various relationships or property values.

Employees must not work on more than 3 projects.

Constraint

```
#Constraint in Terp syntax:
#:Employee rdfs:subClassOf (:works_on max 3 :Project)

:Employee rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty :works_on;
                owl:maxQualifiedCardinality "3"^^xsd:nonNegativeInteger ;
                owl:onClass :Project
              ] .
```

Database A (valid)

```
:Bob a owl:Thing.
```

Database B (valid)

```
:Bob a :Employee ;
        :works_on :MyProject .

:MyProject a :Project .
```

Database C (invalid)

```
:Bob a :Employee ;
        :works_on :MyProject , :MyProjectFoo , :MyProjectBar , :MyProjectBaz .

:MyProject a :Project .

:MyProjectFoo a :Project .

:MyProjectBar a :Project .

:MyProjectBaz a :Project .
```

If an RDF instance `i` is an `Employee`, then `i` must not be related via the property `works_on` to more than 3 instances of `Project`. In A, `Bob` is not known to be an instance of `Employee` so the constraint does not apply and the A is valid. In B, `Bob` is an instance of `Employee` but is known to work on only a single project, namely `MyProject`, so B is valid. In C, `Bob` is related to 4 instances of `Project` via `works_on`.

> **NOTE**
>
> Stardog ICV implements a weak form of the **unique name assumption**, that is, it assumes that things which have different names are, in fact, different things.[37]

Since Stardog ICV uses closed world (instead of open world) semantics,[38] it assumes that the different projects with different names are, in fact, separate projects, which (**in this case**) violates the constraint and makes C invalid.

Departments must have at least 2 employees.

Constraint

```
#Constraint in Terp syntax:
#:Department rdfs:subClassOf (inverse :works_in min 2 :Employee)

:Department rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty [owl:inverseOf :works_in] ;
                owl:minQualifiedCardinality "2"^^xsd:nonNegativeInteger ;
                owl:onClass :Employee
              ] .
```

Database A (valid)

```
[source,sparql]
:MyDepartment a owl:NamedIndividual .
```

Database B (invalid)

```
:MyDepartment a :Department .

:Bob a :Employee ;
        :works_in :MyDepartment .
```

Database C (valid)

```
[source,sparql]
:MyDepartment a :Department .

:Alice a :Employee ;
        :works_in :MyDepartment .

:Bob a :Employee ;
        :works_in :MyDepartment .
```

This constraint says that if an RDF instance `i` is a `Department`, then there should exist at least 2 instances `j` and `k` of class `Employee` which are related to `i` via the property `works_in` (or, equivalently, `i` should be related to them via the inverse of `works_in`). In A, `MyDepartment` is not known to be an instance of `Department` so the constraint does not apply. In B, `MyDepartment` is an instance of `Department` but only one instance of `Employee`, namely `Bob`, is known to work in it, so B is invalid. In C, `MyDepartment` is related to the individuals `Bob` and `Alice`, which are both instances of `Employee` and (again, due to weak Unique Name Assumption in Stardog ICV), are assumed to be distinct, so C is valid.

Managers must manage exactly 1 department.

Constraint

```
#Constraint in Terp syntax:
#:Manager rdfs:subClassOf (:manages exactly 1 :Department)

:Manager rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty :manages ;
                owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger ;
                owl:onClass :Department
              ] .
```

Database A (valid)

```
    Individual: Isabella
```

Database B (invalid)

```
:Isabella a :Manager .
```

Database C (invalid)

```
:Isabella a :Manager ;
        :manages :MyDepartment .
```

Database D (valid)

```
:Isabella a :Manager ;
        :manages :MyDepartment .

:MyDepartment a :Department .
```

Database E (invalid)

```
:Isabella a :Manager ;
        :manages :MyDepartment , :MyDepartment1 .

:MyDepartment a :Department .

:MyDepartment1 a :Department .
```

This constraint says that if an RDF instance `i` is a `Manager`, then it must be related to exactly 1 instance of `Department` via the property `manages`. In A, the individual `Isabella` is not known to be an instance of `Manager` so the constraint does not apply and A is valid. In B, `Isabella` is an instance of `Manager` but is not related to any instances of `Department`, so B is invalid. In C, `Isabella` is related to the individual `MyDepartment` via the property `manages` but `MyDepartment` is not known to be an instance of `Department`, so C is invalid. In D, `Isabella` is related to exactly one instance of `Department`, namely `MyDepartment`, so D is valid. Finally, in E, `Isabella` is related to two (assumed to be) distinct (again, because of weak UNA) instances of `Department`, namely `MyDepartment` and `MyDepartment1`, so E is invalid.

Entities may have no more than one name.

Constraint

```
:name a owl:FunctionalProperty .
```

Database A (valid)

```
:MyDepartment a owl:Thing .
```

Database B (valid)

```
:MyDepartment :name "Human Resources" .
```

Database C (invalid)

```
:MyDepartment :name "Human Resources" , "Legal" .
```

This constraint says that no RDF instance `i` can have more than one assertion via the data property `name`. In A, `MyDepartment` does not have any data property assertions so A is valid. In B, `MyDepartment` has a single assertion via `name`, so the ontology is also valid. In C, `MyDepartment` is related to 2 literals, namely `"Human Resources"` and `"Legal"`, via `name`, so C is invalid.

Property Constraints

These constraints control how instances are related to one another via properties.

The manager of a department must work in that department.

Constraint

```
:manages rdfs:subPropertyOf :works_in .
```

Database A (invalid)

```
:Bob :manages :MyDepartment
```

Database B (valid)

```
:Bob :works_in :MyDepartment ;
        :manages :MyDepartment .
```

This constraint says that if an RDF instance `i` is related to `j` via the property `manages`, then `i` must also be related to `j` va the property `works_in`. In A, `Bob` is related to `MyDepartment` via `manages`, but not via `works_in`, so A is invalid. In B, `Bob` is related to `MyDepartment` via both `manages` and `works_in`, so B is valid.

Department managers must supervise all the department's employees.

Constraint

```
:is_supervisor_of owl:propertyChainAxiom (:manages [owl:inverseOf :works_in]) .
```

Database A (invalid)

```
:Jose :manages :MyDepartment ;
        :is_supervisor_of :Maria .

:Maria :works_in :MyDepartment .

:Diego :works_in :MyDepartment .
```

Database B (valid)

```
:Jose :manages :MyDepartment ;
        :is_supervisor_of :Maria , :Diego .

:Maria :works_in :MyDepartment .

:Diego :works_in :MyDepartment .
```

This constraint says that if an RDF instance `i` is related to `j` via the property `manages` and `k` is related to `j` via the property `works_in`, then `i` must be related to `k` via the property `is_supervisor_of`. In A, `Jose` is related to `MyDepartment` via `manages`, `Diego` is related to `MyDepartment` via `works_in`, but `Jose` is not related to `Diego` via any property, so A is invalid. In B, `Jose` is related to `Maria` and `Diego` --who are both related to `MyDepartment` by way of `works_in` --via the property `is_supervisor_of`, so B is valid.

Complex Constraints

Constrains may be arbitrarily complex and include many conditions.

Employee Constraints

Each employee works on at least one project, or supervises at least one employee that works on at least one project, or manages at least one department.

Constraint

```
#Constraint in Terp syntax:
#how are you not loving Terp by now?!
#:Employee rdfs:subClassOf (:works_on some (:Project or
#(:supervises some (:Employee and (:works_on some :Project))) or (:manages some
:Department)))

:Employee rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty :works_on ;
                owl:someValuesFrom
                        [ owl:unionOf (:Project
                                      [ a owl:Restriction ;
                                        owl:onProperty :supervises ;
                                        owl:someValuesFrom
                                              [ owl:intersectionOf (:Employee
                                                                    [ a
owl:Restriction ;
                                                                      owl:onProperty
:works_on ;

owl:someValuesFrom :Project
                                                                    ])
                                              ]
                                      ]
                                      [ a owl:Restriction ;
                                        owl:onProperty :manages ;
                                        owl:someValuesFrom :Department
                                      ])
                        ]
              ] .
```

Database A (invalid)

```
:Esteban a :Employee .
```

Database B (invalid)

```
:Esteban a :Employee ;
        :supervises :Lucinda .

:Lucinda a :Employee .
```

Database C (valid)

```
:Esteban a :Employee ;
        :supervises :Lucinda .

:Lucinda a :Employee ;
        :works_on :MyProject .

:MyProject a :Project .
```

Database D (valid)

```
:Esteban a :Employee ;
        :manages :MyDepartment .

:MyDepartment a :Department .
```

Database E (valid)

```
:Esteban a :Employee ;
        :manages :MyDepartment ;
        :works_on :MyProject .

:MyDepartment a :Department .

:MyProject a :Project .
```

This constraint says that if an individual `i` is an instance of `Employee`, then at least one of three conditions must be met:

- it is related to an instance of `Project` via the property `works_on`
- it is related to an instance `j` via the property `supervises` and `j` is an instance of `Employee` and is also related to some instance of `Project` via the property `works_on`
- it is related to an instance of `Department` via the property `manages`.

A and B are invalid because none of the conditions are met. C meets the second condition: `Esteban` (who is an `Employee`) is related to `Lucinda` via the property `supervises` whereas `Lucinda` is both an `Employee` and related to `MyProject`, which is a `Project`, via the property `works_on`. D meets the third condition: `Esteban` is related to an instance of `Department`, namely `MyDepartment`, via the property `manages`. Finally, E meets the first and the third conditions because in addition to managing a department `Esteban` is also related an instance of `Project`, namely `MyProject`, via the property `works_on`.

Employees and US government funding

Only employees who are American citizens can work on a project that receives funds from a US government agency.

Constraint

```
#Constraint in Terp syntax:
#:Project and (:receives_funds_from some :US_Government_Agency)) rdfs:subClassOf
#(inverse :works_on only (:Employee and (:nationality value "US")))

[ owl:intersectionOf (:Project
                       [ a owl:Restriction ;
                         owl:onProperty :receives_funds_from ;
                         owl:someValuesFrom :US_Government_Agency
                       ]) .
] rdfs:subClassOf
              [ a owl:Restriction ;
                owl:onProperty [owl:inverseOf :works_on] ;
                owl:allValuesFrom [ owl:intersectionOf (:Employee
                                                        [ a owl:Restriction ;
                                                          owl:hasValue "US" ;
                                                          owl:onProperty :nationality
                                                        ])
                                  ]
              ] .
```

Database A (valid)

```
:MyProject a :Project ;
        :receives_funds_from :NASA .

:NASA a :US_Government_Agency
```

Database B (invalid)

```
:MyProject a :Project ;
        :receives_funds_from :NASA .

:NASA a :US_Government_Agency .

:Andy a :Employee ;
        :works_on :MyProject .
```

Database C (valid)

```
:MyProject a :Project ;
        :receives_funds_from :NASA .

:NASA a :US_Government_Agency .

:Andy a :Employee ;
        :works_on :MyProject ;
        :nationality "US" .
```

Database D (invalid)

```
:MyProject a :Project ;
        :receives_funds_from :NASA .

:NASA a :US_Government_Agency .

:Andy a :Employee ;
        :works_on :MyProject ;
        :nationality "US" .

:Heidi a :Supervisor ;
        :works_on :MyProject ;
        :nationality "US" .
```

Database E (valid)

```
:MyProject a :Project ;
        :receives_funds_from :NASA .

:NASA a :US_Government_Agency .

:Andy a :Employee ;
        :works_on :MyProject ;
        :nationality "US" .

:Heidi a :Supervisor ;
        :works_on :MyProject ;
        :nationality "US" .

:Supervisor rdfs:subClassOf :Employee .
    SubClassOf: Employee
```

This constraint says that if an individual `i` is an instance of `Project` and is related to an instance of `US_Government_Agency` via the property `receives_funds_from`, then any individual `j` which is related to `i` via the property `works_on` must satisfy two conditions:

- it must be an instance of `Employee`
- it must not be related to any literal other than `"US"` via the data property `nationality`.

A is valid because there is no individual related to `MyProject` via `works_on`, so the constraint is trivially satisfied. B is invalid since `Andy` is related to `MyProject` via `works_on`, `MyProject` is an instance of `Project` and is related to an instance of `US_Government_Agency`, that is, `NASA`, via `receives_funds_from`, but `Andy` does not have any data property assertions. C is valid because both conditions are met. D is not valid because `Heidi` violated the first condition: she is related to `MyProject` via `works_on` but is not known to be an instance of `Employee`. Finally, this is fixed in E—by way of a handy OWL axiom—which states that every instance of `Supervisor` is an instance of `Employee`, so `Heidi` is inferred to be an instance of `Employee` and, consequently, E is valid.[39]

If you made it this far, you deserve a drink!

Using ICV Programmatically

Here we describe how to use Stardog ICV via the SNARL APIs. For more information on using SNARL in general, please refer to the chapter on programming Stardog with Java.

There is command-line interface support for many of the operations necessary to using ICV with a Stardog database; please see Administering Stardog for details.

To use ICV in Stardog, one must:

1. create some constraints
2. associate those constraints with a Stardog database

Creating Constraints

`Constraints` can be created using the `ConstraintFactory` which provides methods for creating integrity constraints. `ConstraintFactory` expects your constraints, if they are defined as OWL axioms, as RDF triples (or graph). To aid in authoring constraints in OWL, `ExpressionFactory` is provided for building the RDF equivalent of the OWL axioms of your constraint.

You can also write your constraints in OWL in your favorite editor and load them into the database from your OWL file.

We recommend defining your constraints as OWL axioms, but you are free to define them using SPARQL `SELECT` queries. If you choose to define a constraint using a SPARQL `SELECT` query, please keep in mind that if your query returns results, those are interpreted as the violations of the integrity constraint.

An example of creating a simple constraint using `ExpressionFactory`:

```
URI Product = ValueFactoryImpl.getInstance().createURI("urn:Product");
URI Manufacturer = ValueFactoryImpl.getInstance().createURI("urn:Manufacturer");
URI manufacturedBy = ValueFactoryImpl.getInstance().createURI("urn:manufacturedBy");

// we want to say that a product should be manufactured by a Manufacturer:
Constraint aConstraint = ConstraintFactory.constraint(subClassOf(Product,
                                                                 some(manufacturedBy,
Manufacturer)));
```

Adding Constraints to Stardog

The ICVConnection interface provides programmatic access to the ICV support in Stardog. It provides support for adding, removing and clearing integrity constraints in your database as well as methods for checking whether or not the data is valid; and when it’s not, retrieving the list of violations.

This example shows how to add an integrity constraint to a Stardog database.

```
// We'll start out by creating a validator from our SNARL Connection
ICVConnection aValidator = aConn.as(ICVConnection.class);

// add add a constraint, which must be done in a transaction.
aValidator.addConstraint(aConstraint);
```

Here we show how to add a set of constraints as defined in a local OWL ontology.

```
// We'll start out by creating a validator from our SNARL Connection
ICVConnection aValidator = aConn.as(ICVConnection.class);

// add add a constraint
aValidator.addConstraints()
        .format(RDFFormat.RDFXML)
        .file(new File("myConstraints.owl"));
```

IC Validation

Checking whether or not the contents of a database are valid is easy. Once you have an `ICVConnection` you can simply call its `isValid()` method which will return whether or not the contents of the database are valid with respect to the constraints associated with that database. Similarly, you can provide some `constraints` to the `isValid()` method to see if the data in the database is invalid for those specific constraints; which can be a subset of the constraints associated with the database, or they can be new constraints you are working on.

If the data is invalid for some constraints—either the explicit constraints in your database or a new set of constraints you have authored—you can get some information about what the violation was from the SNARL IC Connection. `ICVConnection.getViolationBindings()` will return the constraints which are violated, and for each constraint, you can get the violations as the set of bindings that satisfied the constraint query. You can turn the bindings into the individuals which are in the violation using `ICV.asIndividuals()`.

ICV and Transactions

In addition to using the ICConnection a data oracle to tell whether or not your data is valid with respect to some constraints, you can also use Stardog's ICV support to protect your database from invalid data by using ICV as a guard within transactions.

When guard mode for ICV is enabled in Stardog, each commit is inspected to ensure that the contents of the database are valid for the set of constraints that have been associated with it. Should someone attempt to commit data which violates one or more of the constraints defined for the database, the commit will fail and the data will not be added/removed from your database.

By default, reasoning is not used when you enable guard mode, however you are free to specify any of the reasoning types supported by Stardog when enabling guard mode. If you have provided a specific reasoning type for guard mode it will be used during validation of the integrity constraints. This means you can author your constraints with the expectation of inference results satisfying a constraint.

```
AdminConnection dbms =
AdminConnectionConfiguration.toEmbeddedServer().credentials("admin",
"admin").connect();

dbms.disk("icvWithGuard")                          // disk db named 'icvWithGuard'
    .set(ICVOptions.ICV_ENABLED, true)          // enable icv guard mode
    .set(ICVOptions.ICV_REASONING_ENABLED, true)          // specify that guard mode
should use reasoning
    .create(new File("data/sp2b_10k.n3"));          // create the db, bulk loading the
file(s) to start

dbms.close();
```

This illustrates how to create a persistent disk database with ICV guard mode and reasoning enabled. Guard mode can also be enabled when the database is created on the CLI.

Terminology

This chapter may make more sense if you read this section on terminology a few times.

ICV, Integrity Constraint Validation

The process of checking whether some Stardog database is valid with respect to some integrity constraints. The result of ICV is a boolean value (true if valid, false if invalid) and, optionally, an `explanation of constraint violations`.

Schema, TBox

A schema (or "terminology box" a.k.a., TBox) is a set of statements that define the relationships between data elements, including property and class names, their relationships, etc. In practical terms, schema statements for a Stardog database are RDF Schema and OWL 2 terms, axioms, and definitions.

Data, ABox

All of the triples in a Stardog database that aren't part of the schema are part of the data (or "assertional box" a.k.a. ABox).

Integrity Constraint

A declarative expression of some rule or constraint which data must conform to in order to be valid. Integrity Constraints are typically domain and application specific. They can be expressed in OWL 2 (any legal syntax), SWRL rules, or (a restricted form of) SPARQL queries.

Constraints

Constraints that have been associated with a Stardog database and which are used to validate the data it contains. Each Stardog may optionally have one and only one set of constraints associated with it.

Closed World Assumption, Closed World Reasoning

Stardog ICV assumes a closed world with respect to data and constraints: that is, it assumes that all relevant data is known to it and included in a database to be validated. It interprets the meaning of Integrity Constraints in light of this assumption; if a constraint says a value `must` be present, the absence of that value is interpreted as a constraint violation and, hence, as invalid data.

Open World Assumption, Open World Reasoning

A legal OWL 2 inference may violate or satisfy an Integrity Constraint in Stardog. In other words, you get to have your cake (OWL as a constraint language) and eat it, too (OWL as modeling or inference language). This means that constraints are applied to a Stardog database `with respect to an OWL 2 profile`.

Monotonicity

OWL is a monotonic language: that means you can never `add` anything to a Stardog database that causes there to be `fewer` legal inferences. Or, put another way, the only way to decrease the number of legal inferences is to `delete` something.

Monotonicity interacts with ICV in the following ways:

1. Adding data to or removing it from a Stardog database may make it invalid.
2. Adding schema statements to or removing them from a Stardog database may make it invalid.
3. Adding new constraints to a Stardog database may make it invalid.
4. Deleting constraints from a Stardog database `cannot make it invalid`.

Programming Stardog

You can program Stardog in Java, over HTTP, JavaScript, Clojure, Groovy, Spring, and .Net.

Sample Code

There's a Github repo of example Java code that you can fork and use as the starting point for your Stardog projects. Feel free to add new examples using pull requests in Github.

Java Programming

In the Network Programming section, we look at how to interact with Stardog over a network via HTTP and SNARL protocol. In this chapter we describe how to program Stardog from Java using SNARL Stardog Native API for the RDF Language, Sesame, and Jena. We prefer SNARL to Sesame to Jena and recommend—all other things being equal—them in that order.

If you're a Spring developer, you might want to read Spring Programming or if you prefer a ORM-style approach, you might want to checkout Empire, an implementation of JPA for RDF that works with Stardog.

Examples

The **best** way to learn to program Stardog with Java is to study the examples:

1. SNARL
2. Sesame bindings
3. Jena bindings
4. SNARL and OWL 2 reasoning
5. SNARL and Connection Pooling
6. SNARL and Searching

We offer some commentary on the interesting parts of these examples below.

Creating & Administering Databases

`AdminConnection` provides simple programmatic access to all administrative functions available in Stardog.

Creating a Database

You can create a basic temporary memory database with Stardog with one line of code:

```
AdminConnection aAdminConnection = AdminConnectionConfiguration.toEmbeddedServer()

.credentials("admin", "admin")
                                                              .connect();

aAdminConnection.createMemory("testConnectionAPI");

// you must always log out of the dbms.
aAdminConnection.close();
```

WARNING It's **crucially important to always clean up connections to the database** by calling `AdminConnection#close().

You can also use the `memory` and `disk` functions to configure and create a database in any way you prefer. These methods return `DatabaseBuilder` objects which you can use to configure the options of the database you'd like to create. Finally, the `create` method takes the list of files to bulk load into the database when you create it and returns a valid `ConnectionConfiguration` which can be used to create new `Connections` to your database.

WARNING It is important to note that you **must** take care to always log out of the server when you are done working with `AdminConnection`.

```
AdminConnection dbms =
AdminConnectionConfiguration.toEmbeddedServer().credentials("admin",
"admin").connect();

dbms.memory("waldoTest")
    .set(SearchOptions.SEARCH_ENABLED, true)
    .create();

dbms.close();
```

This illustrates how to create a temporary memory database named `test` which supports full text search via Searching.

```
AdminConnection dbms =
AdminConnectionConfiguration.toEmbeddedServer().credentials("admin",
"admin").connect();

dbms.disk("icvWithGuard")                          // disk db named 'icvWithGuard'
    .set(ICVOptions.ICV_ENABLED, true)        // enable icv guard mode
    .set(ICVOptions.ICV_REASONING_ENABLED, true)        // specify that guard mode
should use reasoning
    .create(new File("data/sp2b_10k.n3"));        // create the db, bulk loading the
file(s) to start

dbms.close();
```

This illustrates how to create a persistent disk database with ICV guard mode and reasoning enabled. For more information on what the available options for `set` are and what they mean, see the Database Admin section. Also note, Stardog database administration can be performed from the CLI.

Creating a Connection String

As you can see, the `ConnectionConfiguration` in `com.complexible.stardog.api` package class is where the initial action takes place:

```
Connection aConn = ConnectionConfiguration
        .to("noReasoningExampleTest")                    // the name of the db to
connect to
        .credentials("admin", "admin")              // credentials to use while
connecting
        .connect();
```

The `to` method takes a `Database Name` as a string; and then `connect` connects to the database using all specified properties on the configuration. This class and its constructor methods are used for **all** of Stardog's Java APIs: SNARL native Stardog API, Sesame, Jena, as well as HTTP and SNARL protocol. In the latter cases, you must also call `server` and pass it a valid URL to the Stardog server using the HTTP or SNARL protocols.

Without the call to `server` , `ConnectionConfiguration` will attempt to connect to a local, embedded version of the Stardog server. The `Connection` still operates in the standard client-server mode, the only difference is that the server is running in the **same** JVM as your application.

NOTE	Whether using SNARL, Sesame, or Jena, most perhaps all Stardog Java code will use `ConnectionConfiguration` to get a handle on a Stardog database—whether embedded or remote—and, after getting that handle, can use the appropriate API.

See the `ConnectionConfiguration` API docs or How to Make a Connection String for more information.

Managing Security

We discuss the security sytem in Stardog in Security. When logged into the Stardog DBMS you can access all security related features detailed in the security section using any of the core security interfaces for managing users, roles, and permissions.

Using SNARL

In examples 1 and 4 above, you can see how to use SNARL in Java to interact with Stardog. *The SNARL API will give the best performance overall and is the native Stardog API*. It uses some Sesame domain classes but is otherwise a clean-sheet API and implementation.

The SNARL API is fluent with the aim of making code written for Stardog easier to write and easier to maintain. Most objects are easily re-used to make basic tasks with SNARL as simple as possible. We are always interested in feedback on the API, so if you have suggestions or comments, please send them to the mailing list.

Let’s take a closer look at some of the interesting parts of SNARL.

Adding Data

```
aConn.begin();

aConn.add().io()
        .format(RDFFormat.N3)
        .stream(new FileInputStream("data/sp2b_10k.n3"));

Graph aGraph = Graphs.newGraph(ValueFactoryImpl.getInstance()

.createStatement(ValueFactoryImpl.getInstance().createURI("urn:subj"),

ValueFactoryImpl.getInstance().createURI("urn:pred"),

ValueFactoryImpl.getInstance().createURI("urn:obj")));

Resource aContext = ValueFactoryImpl.getInstance().createURI("urn:test:context");

aConn.add().graph(aGraph, aContext);

aConn.commit();
```

You must always enclose changes to a database within a transaction begin and commit or rollback. Changes are local until the transaction is committed or until you try and perform a query operation to inspect the state of the database within the transaction.

By default, RDF added will go into the default context unless specified otherwise. As shown, you can use Adder directly to add statements and graphs to the database; and if you want to add data from a file or input stream, you use the `io`, `format`, and `stream` chain of method invocations.

See the SNARL API Javadocs for all the gory details.

Removing Data

```
// first start a transaction
aConn.begin();
aConn.remove().io()
        .format(RDFFormat.N3)
        .file(new File("data/remove_data.nt"));

// and commit the change
aConn.commit();
```

Let's look at removing data via SNARL; in the example above, you can see that file or stream-based removal is symmetric to file or stream-based addition, i.e., calling `remove` in an `io` chain with a file or stream call. See the SNARL API docs for more details about finer-grained deletes, etc.

Parameterized SPARQL Queries

```
URI aURI = ValueFactoryImpl.getInstance()
                .createURI("http://localhost/publications/articles/Journal1/1940/
Article1");

SelectQuery aQuery = aConn.select("select * where {?s ?p ?o}");

// now we can run this query...but lets set a limit on it since otherwise that'd be
the whole database
aQuery.limit(10);

TupleQueryResult aResult = aQuery.execute();

System.out.println("The first ten results...");

// and do something with the results
while (aResult.hasNext()) {
        System.out.println(aResult.next());
}

// always close your result sets
aResult.close();

// query objects are easily parameterized, we can bind the "s" variable in the
previous query
// with a specific value
aQuery.parameter("s", aURI);

// and remove the limit
```

```
aQuery.limit(SelectQuery.NO_LIMIT);

// now we can re-run the query
aResult = aQuery.execute();

System.out.println("\nNow a particular slice...");

while (aResult.hasNext()) {
        System.out.println(aResult.next());
}

aResult.close();
```

SNARL also lets us parameterize SPARQL queries. We can make a `Query` object by passing a SPARQL query in the constructor. Simple. Obvious.

Next, let's set a limit for the results: `aQuery.limit10`; or if we want no limit, `aQuery.limitQuery.NO_LIMIT`. By default, there is no limit imposed on the query object; we'll use whatever is specified in the query. But you can use limit to override any limit specified in the query, however specifying NO_LIMIT will not remove a limit specified in a query, it will only remove any limit override you've specified, restoring the state to the default of using whatever is in the query.

We can execute that query with `executeSelect` and iterate over the results. We can also rebind the "?s" variable easily: `aQuery.parameter"s", aURI`, which will work for all instances of "?s" in any BGP in the query, and you can specify `null` to remove the binding.

Query objects are re-useable, so you can create one from your original query string and alter bindings, limit, and offset in any way you see fit and re-execute the query to get the updated results.

We **strongly** recommend the use of SNARL's parameterized queries over concatenating strings together in order to build your SPARQL query. This latter approach opens up the possibility for SPARQL injection attacks unless you are very careful in scrubbing your input.[40]

Getter Interface

```java
// The previous query was just getting the statements which the value of aURI is the subject,
// which we can easily do via the getter interface
Iteration<Statement, StardogException> aIter = aConn.get().subject(aURI).iterator();

System.out.println("\nOr you can use a getter to do the same thing...");

while (aIter.hasNext()) {
        System.out.println(aIter.next());
}

// always close your iterations as well...
aIter.close();

// Getter objects are parameterizable like queries, and can be reused

Getter aGetter = aConn.get();

aGetter.predicate(RDF.TYPE);

// calling iterator() on this getter will return all statements which have RDF.TYPE as the predicate
// or we can bind the subject and get a specific type statement...

aGetter.subject(aURI);

// this will return the type triple for aURI as an Iteration
aIter = aGetter.iterator();

System.out.println("\nJust a single statement now...");

while (aIter.hasNext()) {
        System.out.println(aIter.next());
}

aIter.close();

// revert having set the predicate on the getter
aGetter.predicate(null);

// we can also get the results as a graph:
aGraph = aGetter.graph();
```

```
System.out.println("\nFinally, the same results as earlier, but as a graph...");
GraphIO.writeGraph(aGraph, new OutputStreamWriter(System.out), RDFFormat.TURTLE);
```

SNARL also supports some sugar for the classic statement-level `getSPO` --scars, anyone?--interactions. We ask in the first line of the snippet above for an iterator over the Stardog connection, based on `aURI` in the subject position. Then a while-loop, as one might expect… You can also parameterize ``Getter`s by binding different positions of the `Getter`` which acts like a kind of RDF statement filter—and then iterating as usual.

> **NOTE** the `aIter.close` which is important for Stardog databases to avoid memory leaks. If you need to materialize the iterator as a graph, you can do that by calling `graph`.

The snippet doesn't show `object` or `context` parameters on a `Getter`, but those work, too, in the obvious way.

Reasoning

Stardog supports query-time reasoning using a query rewriting technique. In short, when reasoning is requested, a query is automatically rewritten to **n** queries, which are then executed. As we discuss below in Connection Pooling, reasoning is enabled at the `Connection` layer and then any queries executed over that connection are executed with reasoning enabled; you don't need to do anything up front when you create your database if you want to use reasoning.

```
ReasoningConnection aReasoningConn = ConnectionConfiguration
        .to("reasoningExampleTest")
        .credentials("admin", "admin")
        .reasoning(true)
        .connect()
        .as(ReasoningConnection.class);
```

In this code example, you can see that it's trivial to enable reasoning for a `Connection`: simply call `reasoning` with `true` passed in.

Search

Stardog's search system can be used from Java. The fluent Java API for searching in SNARL looks a lot like the other search interfaces: We create a `Searcher` instance with a fluent constructor: `limit` sets a limit on the results; `query` contains the search query, and `threshold` sets a minimum threshold for the results.

```
Searcher aSearch = aSearchConn.search()
        .limit(50)                 // as before we only want the top fifty results
        .query("mac")                // our search term
        .threshold(0.5);        // Since Waldo is implemented over lucene, we can
also specify a min threshold for our results

SearchResults aSearchResults = aSearch.search();

// and now we can just iterate over the search results

Iteration<SearchResult, QueryEvaluationException> resultIt =
aSearchResults.iteration();

System.out.println("\nAPI results: ");
while (resultIt.hasNext()) {
        SearchResult aHit = resultIt.next();

        System.out.println(aHit.getHit() + " with a score of: " + aHit.getScore());
}

// don't forget to close your iteration!
resultIt.close();

// we can also re-use the searcher if we want to find the next set of results...

aSearch.offset(50); // we already found the first fifty, so lets grab the next set

aSearchResults = aSearch.search();

// we can now check the next page of search results!
```

Then we call the `search` method of our `Searcher` instance and iterate over the results i.e., `SearchResults`. Last, we can use `offset` on an existing `Searcher` to grab another page of results.

Stardog also supports performing searches over the full-text index **within** a SPARQL query via the LARQ SPARQL syntax. This provides a powerful mechanism for querying both your RDF index and full-text index at the same time while also giving you a more performant option to the SPARQL `regex` filter.

User-defined Lucene Analyzer

Stardog's Semantic Search capability uses Lucene's default text analyzer, which may not be ideal for your data or application. You can implement a custom analyzer that Stardog will use by implementing `org.apache.lucene.analysis.Analyzer`. That lets you customize Stardog to support different natural languages, geospatial indexing, domain-specific stop word lists, etc.

See Custom Analyzers in the stardog-examples Github repo for a complete description of the API, registry, sample code, etc.

User-defined Functions and Aggregates

Stardog may be extended via Function and Aggregate extensibility APIs, which are fully documented, including sample code, in the stardog-examples Github repo section about function extensibility.

In short you can extend Stardog's SPARQL query evaluation with custom functions and aggregates easily. Function extensibility corresponds to built-in expressions used in `FILTER`, `BIND` and `SELECT` expressions, as well as aggregate operators in a SPARQL query like `COUNT` or `SAMPLE`.

SNARL Connection Views

SNARL `Connections` support obtaining a specified type of `Connection`. This lets you extend and enhance the features available to a `Connection` while maintaining the standard, simple Connection API. The Connection `as` method takes as a parameter the interface, which must be a sub-type of a `Connection`, that you would like to use. `as` will either return the `Connection` as the view you've specified, or it will throw an exception if the view could not be obtained for some reason.

An example of obtaining an instance of a `SearchConnection` to use Stardog's full-text search support would look like this:

```
SearchConnection aSearchConn = aConn.as(SearchConnection.class);
```

SNARL API Docs

Please see SNARL API docs for more information.

Using Sesame

Stardog supports the Sesame API; thus, for the most part, using Stardog and Sesame is not much different from using Sesame with other RDF databases. There are, however, at least two differences worth pointing out.

Wrapping connections with StardogRepository

```
// Create a Sesame Repository from a Stardog ConnectionConfiguration.  The
configuration will be used
// when creating new RepositoryConnections
Repository aRepo = new StardogRepository(ConnectionConfiguration
                                                 .to("testSesame")
                                                 .credentials("admin", "admin"));

// init the repo
aRepo.initialize();

// now you can use it like a normal Sesame Repository
RepositoryConnection aRepoConn = aRepo.getConnection();

// always best to turn off auto commit
aRepoConn.setAutoCommit(false);
```

As you can see from the code snippet, once you've created a `ConnectionConfiguration` with all the details for connecting to a Stardog database, you can wrap that in a `StardogRepository` which is a Stardog-specific implementation of the Sesame `Repository` interface. At this point, you can use the resulting `Repository` like any other Sesame `Repository` implementation. Each time you call `Repository.getConnection`, your original `ConnectionConfiguration` will be used to spawn a new connection to the database.

Autocommit

Stardog’s `RepositoryConnection` implementation will, by default, disable `autoCommit` status. When enabled, every single statement added or deleted via the `Connection` will incur the cost of a transaction, which is too heavyweight for most use cases. You can enable `autoCommit` and it will work as expected; but **we recommend leaving it disabled**.

Using Jena

Stardog supports Jena via a Sesame-Jena bridge, so it’s got more overhead than Sesame or SNARL. YMMV. There two points in the Jena example to emphasize.

Init in Jena

```
// obtain a Jena model for the specified stardog database connection.  Just creating
an in-memory
// database; this is roughly equivalent to ModelFactory.createDefaultModel.
Model aModel = SDJenaFactory.createModel(aConn);
```

The initialization in Jena is a bit different from either SNARL or Sesame; you can get a Jena `Model` instance by passing the `Connection` instance returned by `ConnectionConfiguration` to the Stardog factory, `SDJenaFactory`.

Add in Jena

```
// start a transaction before adding the data.  This is not required,
// but it is faster to group the entire add into a single transaction rather
// than rely on the auto commit of the underlying stardog connection.
aModel.begin();

// read data into the model.  note, this will add statement at a time.
// Bulk loading needs to be performed directly with the BulkUpdateHandler provided
// by the underlying graph, or by reading in files in RDF/XML format, which uses the
// bulk loader natively.  Alternatively, you can load data into the Stardog
// database using SNARL, or via the command line client.
aModel.getReader("N3").read(aModel, new FileInputStream("data/sp2b_10k.n3"), "");
```

```
// done!
aModel.commit();
```

Jena also wants to add data to a `Model` one statement at a time, which can be less than ideal. To work around this restriction, we recommend adding data to a `Model` in a single Stardog transaction, which is initiated with `aModel.begin`. Then to read data into the model, we recommend using RDF/XML, since that triggers the `BulkUpdateHandler` in Jena or grab a `BulkUpdateHandler` directly from the underlying Jena graph.

The other options include using the Stardog CLI client to bulk load a Stardog database or to use SNARL for loading and then switch to Jena for other operations, processing, query, etc.

Client-Server Stardog

Using Stardog from Java in either embedded or client-server mode is **very similar**--the only visible difference is the use of `url` in a `ConnectionConfiguration`: when it's present, we're in client-server model; else, we're in embedded mode.

That's a good and a bad thing: it's good because the code is symmetric and uniform. It's bad because it can make reasoning about performance difficult, i.e., it's not entirely clear in client-server mode which operations trigger or don't trigger a round trip with the server and, thus, which may be more expensive than they are in embedded mode.

In client-server mode, **everything triggers a round trip** with these exceptions:

- closing a connection outside a transaction
- any parameterizations or other of a query or getter instance
- any database state mutations in a transaction that don't need to be immediately visible to the transaction; that is, changes are sent to the server only when they are required, on commit, or on any query or read operation that needs to have the accurate up-to-date state of the data within the transaction.

Stardog generally tries to be as lazy as possible; but in client-server mode, since state is maintained on the client, there are fewer chances to be lazy and more interactions with the server.

Connection Pooling

Stardog supports connection pools for SNARL `Connection` objects for efficiency and programmer sanity. Here’s how they work:

```
Server aServer = Stardog
        .buildServer()
        .bind(SNARLProtocolConstants.EMBEDDED_ADDRESS)
        .start();

// First create a temporary database to use (if there is one already, drop it first)
AdminConnection aAdminConnection =
AdminConnectionConfiguration.toEmbeddedServer().credentials("admin",
"admin").connect();
if (aAdminConnection.list().contains("testConnectionPool")) {
        aAdminConnection.drop("testConnectionPool");
}
aAdminConnection.createMemory("testConnectionPool");
aAdminConnection.close();

// Now, we need a configuration object for our connections, this is all the
information about
// the database that we want to connect to.
ConnectionConfiguration aConnConfig = ConnectionConfiguration
        .to("testConnectionPool")
        .credentials("admin", "admin");

// We want to create a pool over these objects.  See the javadoc for
ConnectionPoolConfig for
// more information on the options and information on the defaults.
ConnectionPoolConfig aConfig = ConnectionPoolConfig
        .using(aConnConfig)                                // use my connection
configuration to spawn new connections
        .minPool(10)                                        // the number of objects
to start my pool with
        .maxPool(1000)                                        // the maximum number
of objects that can be in the pool (leased or idle)
```

```
        .expiration(1, TimeUnit.HOURS)                        // Connections can
expire after being idle for 1 hr.
        .blockAtCapacity(1, TimeUnit.MINUTES);                // I want obtain to
block for at most 1 min while trying to obtain a connection.

// now i can create my actual connection pool
ConnectionPool aPool = aConfig.create();

// if I want a connection object...
Connection aConn = aPool.obtain();

// now I can feel free to use the connection object as usual...

// and when I'm done with it, instead of closing the connection, I want to return it
to the pool instead.
aPool.release(aConn);

// and when I'm done with the pool, shut it down!
aPool.shutdown();

// you MUST stop the server if you've started it!
aServer.stop();
```

Per standard practice, we first initialize security and grab a connection, in this case to the `testConnectionPool` database. Then we setup a `ConnectionPoolConfig`, using its fluent API, which establishes the parameters of the pool:

`using` Sets which ConnectionConfiguration we want to pool; this is what is used to actually create the connections.

`minPool`, `maxPool` Establishes min and max pooled objects; max pooled objects includes both leased and idled objects.

`expiration` Sets the idle life of objects; in this case, the pool reclaims objects idled for 1 hour.

`blockAtCapacity`	Sets the max time in minutes that we'll block waiting for an object when there aren't any idle ones in the pool.

Whew! Next we can `create` the pool using this `ConnectionPoolConfig` thing.

Finally, we call `obtain` on the `ConnectionPool` when we need a new one. And when we're done with it, we return it to the pool so it can be re-used, by calling `release`. When we're done, we `shutdown` the pool.

Since reasoning in Stardog is enabled per `Connection`, you can create two pools: one with reasoning connections, one with non-reasoning connections; and then use the one you need to have reasoning **per query**; never pay for more than you need.

API Deprecation

Methods and classes in SNARL API that are marked with the `com.google.common.annotations.Beta` are subject to change or removal in any release. We are using this annotation to denote new or experimental features, the behavior or signature of which may change significantly before it's out of "beta".

We will otherwise attempt to keep the public APIs as stable as possible, and methods will be marked with the standard `@Deprecated` annotation for a least one full revision cycle before their removal from the SNARL API. See Compatibility Policies for more information about API stability.

Anything marked `@VisibleForTesting` is just that, visible as a consequence of test case requirements; don't write any important code that depends on functions with this annotation.

Using Maven

As of Stardog 3.0, we support Maven for both client and server JARs. The following table summarizes the type of dependencies that you will have to include in your project, depending on whether the project is a Stardog client, or server, or both. Additionally, you can also include the Jena or Sesame bindings if you would like to use them in your project. The Stardog dependency list below follows the Gradle convention and is of the form: `groupId:artifactId:VERSION`. Versions 3.0 and higher are supported.

7. Table of client type dependencies

Type	Stardog Dependency
snarl client	`com.complexible.stardog:client-snarl:VERSION`
http client	`com.complexible.stardog:client-http:VERSION`
server	`com.complexible.stardog:server:VERSION`
sesame	`com.complexible.stardog.sesame:stardog-sesame-core:VERSION`
jena	`com.complexible.stardog.jena:stardog-jena:VERSION`

You can see an example of their usage in our examples repository on Github.

Public Maven Repo

The public Maven repository for the current Stardog release is http://maven.stardog.com. To get started, you need to add the following endpoint to your preferred build system, e.g. in your Gradle build script:

```
repositories {
  maven {
    url "http://maven.stardog.com"
  }
}
```

Similarly, if you’re using Maven you’ll need to add the following to your Maven `pom.xml` :

```
<repositories>
   <repository>
     <id>stardog-public</id>
     <url>http://maven.stardog.com</url>
   </repository>
</repositories>
```

NOTE If you’re using Maven as your build tool, then `client-snarl` , `client-http` , and `server` dependencies require that you specify the packaging type. For example,

```
<dependency>
  <groupId>com.complexible.stardog</groupId>
  <artifactId>client-snarl</artifactId>
  <version>$VERSION</version>
  <type>pom</type>
</dependency>
```

Private Maven Repo

CUSTOMER ACCESS

This feature or service is available to Stardog customers. For information about licensing, please email us.

For access to nightly builds, priority bug fixes, priority feature access, hot fixes, etc. Enterprise Premium Support customers have access to their own private Maven repository that is linked to our internal development repository. We provide a private repository which you can either proxy from your preferred Maven repository manager—e.g. Artifactory or Nexus—or add the private endpoint to your build script.

Connecting to Your Private Maven Repo

Similar to our public Maven repo, we will provide you with a private URL and credentials to your private repo, which you will refer to in your Gradle build script like this:

```
repositories {
  maven {
    url $yourPrivateUrl
       credentials {
          username $yourUsername
          password $yourPassword
        }
    }
}
```

Or if you're using Maven, add the following to your `pom.xml`:

```
<repositories>
   <repository>
      <id>stardog-private</id>
      <url>$yourPrivateUrl</url>
   </repository>
</repositories>
```

Then in your `~./m2/settings.xml` add:

```
<settings>
  <servers>
    <server>
      <id>stardog-private</id>
      <username>$yourUsername</username>
      <password>$yourPassword</password>
    </server>
  </servers>
</settings>
```

Network Programming

In the Java Programming section, we consider interacting with Stardog programatically from a Java program. In this section we consider interacting with Stardog over HTTP. In some use cases or deployment scenarios, it may be necessary to interact with or control Stardog remotely over an IP-based network.

Stardog supports SPARQL 1.0 HTTP Protocol; the SPARQL 1.1 Graph Store HTTP Protocol; the Stardog HTTP Protocol; and SNARL, an RPC-style protocol based on Google Protocol Buffers.

SPARQL Protocol

Stardog supports the standard SPARQL Protocol HTTP bindings, as well as additional functionality via HTTP. Stardog also supports SPARQL 1.1's Service Description format. See the spec if you want details.

Stardog HTTP Protocol

The Stardog HTTP Protocol supports SPARQL Protocol 1.1 and additional resource representations and capabilities. The Stardog HTTP API v4 is also available on Apiary: http://docs.stardog.apiary.io/. The Stardog Linked Data API (aka "Annex") is also documented on Apiary: http://docs.annex.apiary.io/.

Generating URLs

If you are running the HTTP server at

```
http://localhost:12345/
```

To form the URI of a particular Stardog Database, the Database Short Name is the first URL path segment appended to the deployment URI. For example, for the Database called `cytwombly`, deployed in the above example HTTP server, the Database Network Name might be

```
http://localhost:12345/cytwombly
```

All the resources related to this database are identified by URL path segments relative to the Database Network Name; hence:

```
http://localhost:12345/cytwombly/size
```

In what follows, we use URI Template notation to parameterize the actual request URLs, thus: `/{db}/size`.

We also abuse notation to show the permissible HTTP request types and default MIME types in the following way: `REQ | REQ /resource/identifier → mime_type | mime_type`. In a few cases, we use `void` as short hand for the case where there is a response code but the response body may be empty.

HTTP Headers: Content-Type & Accept

All HTTP requests that are mutative (add or remove) must include a valid `Content-Type` header set to the MIME type of the request body, where "valid" is a valid MIME type for N-Triples, Trig, Trix, Turtle, NQuads, JSON-LD, or RDF/XML:

RDF/XML	`application/rdf+xml`
Turtle	`application/x-turtle` or `text/turtle`
N-Triples	`text/plain`
TriG	`application/x-trig`
TriX	`application/trix`

NQuads	`text/x-nquads`
JSON-LD	`application/ld+json`

SPARQL `CONSTRUCT` queries must also include a `Accept` header set to one of these RDF serialization types.

When issuing a `SELECT` query the `Accept` header should be set to one of the valid MIME types for `SELECT` results:

SPARQL XML Results Format	`application/sparql-results+xml`
SPARQL JSON Results Format	`application/sparql-results+json`
SPARQL Boolean Results	`text/boolean`
SPARQL Binary Results	`application/x-binary-rdf-results-table`

Response Codes

Stardog uses the following HTTP response codes:

`200` Operation has succeeded.

`202` Operation was received successfully and will be processed shortly.

`400` Indicates parse errors or that the transaction identifier specified for an operation is invalid or does not correspond to a known transaction.

`401` Request is unauthorized.

`403` User attempting to perform the operation does not exist, their username or password is invalid, or they do not have the proper credentials to perform the action.

`404` A resource involved in the request—for example the database or transaction—does not exist.

`409` A conflict for some database operations; for example, creating a database that already exists.

`500` A unspecified failure in some internal operation…Call your office, Senator!

There are also Stardog-specific error codes in the `SD-Error-Code` header in the response from the server. These can be used to further clarify the reason for the failure on the server, especially in cases where it could be ambiguous. For example, if you received a `404` from the server trying to commit a transaction denoted by the path `/myDb/transaction/commit/293845klf9f934`…it's probably not clear what is missing: it's either the transaction or the database. In this case, the value of the `SD-Error-Code` header will clarify.

The enumeration of `SD-Error-Code` values and their meanings are as follows:

`0` Authentication error

`1` Authorization error

`2` Query evaluation error

`3`	Query contained parse errors
`4`	Query is unknown
`5`	Transaction not found
`6`	Database not found
`7`	Database already exists
`8`	Database name is invalid
`9`	Resource (user, role, etc) already exists
`10`	Invalid connection parameter(s)
`11`	Invalid database state for the request
`12`	Resource in use
`13`	Resource not found
`14`	Operation not supported by the server
`15`	Password specified in the request was invalid

In cases of error, the message body of the result will include any error information provided by the server to indicate the cause of the error.

Stardog Resources

To interact with Stardog over HTTP, use the following resource representations, HTTP response codes, and resource identifiers.

A Stardog Database

```
GET /{db} → void
```

Returns a representation of the database. As of Stardog 3.1.4, this is merely a placeholder; in a later release, this resource will serve the web console where the database can be interacted with in a browser.

Database Size

```
GET /{db}/size → text/plain
```

Returns the number of RDF triples in the database.

Query Evaluation

```
GET | POST /{db}/query
```

The SPARQL endpoint for the database. The valid Accept types are listed above in the HTTP Headers section.

To issue SPARQL queries with reasoning over HTTP, see Using Reasoning.

SPARQL update

```
GET | POST /{db}/update → text/boolean
```

The SPARQL endpoint for updating the database with SPARQL Update. The valid Accept types are `application/sparql-update` `or application/x-www-form-urlencoded`. Response is the result of the update operation as text, eg `true` or `false`.

Query Plan

```
GET | POST /{db}/explain → text/plain
```

Returns the explanation for the execution of a query, i.e., a query plan. All the same arguments as for Query Evaluation are legal here; but the only MIME type for the Query Plan resource is `text/plain`.

Transaction Begin

```
POST /{db}/transaction/begin → text/plain
```

Returns a transaction identifier resource as `text/plain`, which is likely to be deprecated in a future release in favor of a hypertext format. `POST` to begin a transaction accepts neither body nor arguments.

Transaction Security Considerations

> **WARNING** Stardog's implementation of transactions with HTTP is vulnerable to man-in-the-middle attacks, which could be used to violate Stardog's isolation guarantee (among other nasty side effects).

Stardog's transaction identifiers are 64-bit GUIDs and, thus, pretty hard to **guess**; but if you can grab a response in-flight, you can steal the transaction identifier if basic access auth or RFC 2069 digest auth is in use. **You've been warned.**

In a future release, Stardog will use RFC 2617 HTTP Digest Authentication, which is less vulnerable to various attacks and will never ask a client to use a different authentication type, which should lessen the likelihood of MitM attacks for properly restricted Stardog clients—that is, a Stardog client that treats any request by a proxy server or origin server (i.e., Stardog) to use basic access auth or RFC 2069 digest auth as a MitM attack. See RFC 2617 for more information.

Transaction Commit

```
POST /{db}/transaction/commit/{txId} → void | text/plain
```

Returns a representation of the committed transaction; `200` means the commit was successful. Otherwise a `500` error indicates the commit failed and the text returned in the result is the failure message.

As you might expect, failed commits exit cleanly, rolling back any changes that were made to the database.

Transaction Rollback

```
POST /{db}/transaction/rollback/{txId} → void | text/plain
```

Returns a representation of the transaction after it's been rolled back. `200` means the rollback was successful, otherwise `500` indicates the rollback failed and the text returned in the result is the failure message.

Querying (Transactionally)

```
GET | POST /{db}/{txId}/query
```

Returns a representation of a query executed within the `txId` transaction. Queries within transactions will be slower as extra processing is required to make the changes visible to the query. Again, the valid Accept types are listed above in the `HTTP Headers` section.

```
GET | POST /{db}/{txId}/update → text/boolean
```

The SPARQL endpoint for updating the database with SPARQL Update. Update queries are executed within the specified transaction `txId` and are **not** atomic operations as with the normal SPARQL update endpoint. The updates are executed when the transaction is committed like any other change. The valid Accept types are `application/sparql-update` `or` `application/x-www-form-urlencoded`. Response is the result of the update operation as text, eg `true` or `false`.

Adding Data (Transactionally)

```
POST /{db}/{txId}/add → void | text/plain
```

Returns a representation of data added to the database of the specified transaction. Accepts an optional parameter, `graph-uri`, which specifies the named graph the data should be added to. If a named graph is not specified, the data is added to the default (i.e., unnamed) context. The response codes are `200` for success and `500` for failure.

Deleting Data (Transactionally)

```
POST /{db}/{txId}/remove → void | text/plain
```

Returns a representation of data removed from the database within the specified transaction. Also accepts `graph-uri` with the analogous meaning as above--Adding Data (Transactionally). Response codes are also the same.

Clear Database

```
POST /{db}/{txId}/clear → void | text/plain
```

Removes all data from the database within the context of the transaction. `200` indicates success; `500` indicates an error. Also takes an optional parameter, `graph-uri`, which removes data from a named graph. To clear only the default graph, pass `DEFAULT` as the value of `graph-uri`.

Export Database

```
GET /{db}/export → RDF
```

Exports the default graph in the database in Turtle format. Also takes an optional parameter, `graph-uri`, which selects a named graph to export. The valid Accept types are the ones defined above in HTTP Headers for RDF Formats.

Explanation of Inferences

```
POST /{db}/reasoning/explain → RDF
POST /{db}/reasoning/{txId}/explain → RDF
```

Returns the explanation of the axiom which is in the body of the `POST` request. The request takes the axioms in any supported RDF format and returns the explanation for why that axiom was inferred as Turtle.

Explanation of Inconsistency

```
GET | POST /{db}/reasoning/explain/inconsistency → RDF
```

If the database is logically inconsistent, this returns an explanation for the inconsistency.

Consistency

```
GET | POST /{db}/reasoning/consistency → text/boolean
```

Returns whether or not the database is consistent w.r.t to the TBox.

Listing Integrity Constraints

```
GET /{db}/icv → RDF
```

Returns the integrity constraints for the specified database serialized in any supported RDF format.

Adding Integrity Constraints

```
POST /{db}/icv/add
```

Accepts a set of valid Integrity constraints serialized in any RDF format supported by Stardog and adds them to the database in an atomic action. 200 return code indicates the constraints were added successfully, 500 indicates that the constraints were not valid or unable to be added.

Removing Integrity Constraints

```
POST /{db}/icv/remove
```

Accepts a set of valid Integrity constraints serialized in any RDF format supported by Stardog and removes them from the database in a single atomic action. `200` indicates the constraints were successfully remove; `500` indicates an error.

Clearing Integrity Constraints

```
POST /{db}/icv/clear
```

Drops **all** integrity constraints for a database. `200` indicates all constraints were successfully dropped; `500` indicates an error.

Converting Constraints to SPARQL Queries

```
POST /{db}/icv/convert
```

The body of the `POST` is a single integrity constraint, serialized in any supported RDF format, with `Content-type` set appropriately. Returns either a `text/plain` result containing a single SPARQL query; or it returns `400` if more than one constraint was included in the input.

Admin Resources

To administer Stardog over HTTP, use the following resource representations, HTTP response codes, and resource identifiers.

List databases

```
GET /admin/databases → application/json
```

Lists all the databases available.

Output JSON example:

```
{ "databases" : ["testdb", "exampledb"] }
```

Copy a database

```
PUT /admin/databases/{db}/copy?to={db_copy}
```

Copies a database `db` to another specified `db_copy` .

Create a new database

```
POST /admin/databases
```

Creates a new database; expects a multipart request with a JSON specifying database name, options and filenames followed by (optional) file contents as a multipart `POST` request.

Expected input (`application/json`):

```
    {
      "dbname" : "testDb",
      "options" : {
        "icv.active.graphs" : "http://graph, http://another",
        "search.enabled" : true,
        ...
      },
      "files" : [{ "filename":"fileX.ttl", "context":"some:context" }, ...]
    }
```

Drop an existing database

```
DELETE /admin/databases/{db}
```

Drops an existing database `db` and all the information that it contains. Goodbye Callahan!

Migrate an existing database

```
PUT /admin/databases/{db}/migrate
```

Migrates the existing content of a legacy database to new format.

Optimize an existing database

```
PUT /admin/databases/{db}/optimize
```

Optimize an existing database.

Sets an existing database online.

```
PUT /admin/databases/{db}/online
```

Request message to set an existing database database online.

Sets an existing database offline.

```
PUT /admin/databases/{db}/offline
```

Request message to set an existing database offline; receives optionally a JSON input to specify a timeout for the offline operation. When not specified, defaults to 3 minutes as the timeout; the timeout should be provided in **milliseconds**. The timeout is the amount of time the database will wait for existing connections to complete before going offline. This will allow open transaction to commit/rollback, open queries to complete, etc. After the timeout has expired, all remaining open connections are closed and the database goes offline.

Optional input (`application/json`):

```
{ "timeout" : timeout_in_ms}
```

Set option values to an existing database.

```
POST /admin/databases/{kb}/options
```

Set options in the database passed through a JSON object specification, i.e. JSON Request for option values. Database options can be found here.

Expected input (`application/json`):

```
    {
      "database.name" : "DB_NAME",
      "icv.enabled" : true | false,
      "search.enabled" : true | false,
      ...
    }
```

Get option values of an existing database.

```
PUT /admin/databases/{kb}/options → application/json
```

Retrieves a set of options passed via a JSON object. The JSON input has empty values for each key, but will be filled with the option values in the database in the output.

Expected input:

```
    {
      "database.name" : ...,
      "icv.enabled" : ...,
      "search.enabled" : ...,
      ...
    }
```

Output JSON example:

```
    {
      "database.name" : "testdb",
      "icv.enabled" : true,
      "search.enabled" : true,
      ...
    }
```

Add a new user to the system.

```
POST /admin/users
```

Adds a new user to the system; allows a configuration option for superuser as a JSON object. Superuser configuration is set as default to false. The password **must** be provided for the user.

Expected input:

```
    {
      "username"  : "bob",
      "superuser" : true | false
      "password"  : "passwd"
    }
```

Change user password.

```
PUT /admin/users/{user}/pwd
```

Changes user’s password in the system. Receives input of new password as a JSON Object.

Expected input:

```
{"password" : "xxxxx"}
```

Check if user is enabled.

```
GET /admin/users/{user}/enabled → application/json
```

Verifies if user is enabled in the system.

Output JSON example:

```
    {
      "enabled": true
    }
```

Check if user is superuser.

```
GET /admin/users/{user}/superuser → application/json
```

Verifies if the user is a superuser:

```
    {
      "superuser": true
    }
```

Listing users.

```
GET /admin/users → application/json
```

Retrieves a list of users.

Output JSON example:

```
    {
      "users": ["anonymous", "admin"]
    }
```

Listing user roles.

```
GET /admin/users/{user}/roles → application/json
```

Retrieves the list of the roles assigned to user.

Output JSON example:

```
    {
      "roles": ["reader"]
    }
```

Deleting users.

```
DELETE /admin/users/{user}
```

Removes a user from the system.

Enabling users.

```
PUT /admin/users/{user}/enabled
```

Enables a user in the system; expects a JSON object in the following format:

```
    {
      "enabled" : true
    }
```

Setting user roles.

```
PUT /admin/users/{user}/roles
```

Sets roles for a given user; expects a JSON object specifying the roles for the user in the following format:

```
    {
      "roles" : ["reader","secTestDb-full"]
    }
```

Adding new roles.

```
POST /admin/roles
```

Adds the new role to the system.

Expected input:

```
 {
   "rolename" : ""
 }
```

Listing roles.

```
GET /admin/roles → application/json
```

Retrieves the list of roles registered in the system.

Output JSON example:

```
{
   "roles": ["reader"]
 }
```

Listing users with a specified role.

```
GET /admin/roles/{role}/users → application/json
```

Retrieves users that have the role assigned.

Output JSON example:

```
{
   "users": ["anonymous"]
}
```

Deleting roles.

```
DELETE /admin/roles/{role}?force={force}
```

Deletes an existing role from the system; the force parameter is a boolean flag which indicates if the delete call for the role must be forced.

Assigning permissions to roles.

```
PUT /admin/permissions/role/{role}
```

Creates a new permission for a given role over a specified resource; expects input JSON Object in the following format:

```
{
   "action" : "read" | "write" | "create" | "delete" | "revoke" | "execute" | "grant"
| "*",
   "resource_type" : "user" | "role" | "db" | "named-graph" | "metadata" | "admin" |
"icv-constraints" | "*",
   "resource" : ""
}
```

Assigning permissions to users.

```
PUT /admin/permissions/user/{user}
```

Creates a new permission for a given user over a specified resource; expects input JSON Object in the following format:

```
{
   "action" : "read" | "write" | "create" | "delete" | "revoke" | "execute" | "grant"
| "*",
   "resource_type" : "user" | "role" | "db" | "named-graph" | "metadata" | "admin" |
"icv-constraints" | "*",
   "resource" : ""
 }
```

Deleting permissions from roles.

```
POST /admin/permissions/role/{role}/delete
```

Deletes a permission for a given role over a specified resource; expects input JSON Object in the following format:

```
 {
    "action" : "read" | "write" | "create" | "delete" | "revoke" | "execute" |
"grant" | "*",
    "resource_type" : "user" | "role" | "db" | "named-graph" | "metadata" | "admin" |
"icv-constraints" | "*",
    "resource" : ""
 }
```

Deleting permissions from users.

```
POST /admin/permissions/user/{user}/delete
```

Deletes a permission for a given user over a specified resource; expects input JSON Object in the following format:

```
{
   "action" : "read" | "write" | "create" | "delete" | "revoke" | "execute" | "grant"
| "*",
   "resource_type" : "user" | "role" | "db" | "named-graph" | "metadata" | "admin" |
"icv-constraints" | "*",
   "resource" : ""
}
```

Listing role permissions.

```
GET /admin/permissions/role/{role} → application/json
```

Retrieves permissions assigned to the role.

Output JSON example:

```
 {
   "permissions": ["stardog:read:*"]
 }
```

Listing user permissions.

```
GET /admin/permissions/user/{user} → application/json
```

Retrieves permissions assigned to the user.

Output JSON example:

```
 {
   "permissions": ["stardog:read:*"]
 }
```

Listing user effective permissions.

```
GET /admin/permissions/effective/user/{user} → application/json
```

Retrieves effective permissions assigned to the user.

Output JSON example:

```
    {
      "permissions": ["stardog:*"]
    }
```

Shutdown server.

```
POST /admin/shutdown
```

Shuts down the Stardog Server. If successful, returns a `202` to indicate that the request was received and that the server will be shut down shortly.

Query Version Metadata

```
GET | POST /{db}/vcs/query
```

Issue a query over the version history metadata using SPARQL. Method has the same arguments and outputs as the normal query method of a database.

Versioned Commit

```
POST /{db}/vcs/{tid}/commit_msg
```

Input example:

```
This is the commit message
```

Accepts a commit message in the body of the request and performs a VCS commit of the specified transaction

Create Tag

```
POST /{db}/vcs/tags/create
```

Input example:

```
"f09c0e02350627480839da4661b8e9cbd70f6372", "This is the commit message"
```

Create a tag from the given revision id with the specified commit message.

Delete Tag

```
POST /{db}/vcs/tags/delete
```

Input example:

```
"f09c0e02350627480839da4661b8e9cbd70f6372"
```

Delete the tag with the given revision.

Revert to Tag

```
POST /{db}/vcs/revert
```

Input example:

```
"f09c0e02350627480839da4661b8e9cbd70f6372",
"893220fba7910792084dd85207db94292886c4d7", "This is the revert message"
```

Perform a revert of a revision to the specified revision with the given commit message.

Javascript Programming

The documentation for stardog.js is available at the stardog.js site; source code is available on Github and npm.

stardog.js

This framework wraps all the functionality of a client for the Stardog DBMS and provides access to a full set of functions such as executing SPARQL Queries, administration tasks on Stardog, and the use of the Reasoning API.

The implementation uses the HTTP protocol, since most of Stardog functionality is available using this protocol. For more information, see Network Programming.

The framework is currently supported for node.js and the browser, including test cases for both environments. You'll also need npm and bower to run the test cases and install the dependencies in node.js & the browser respectively.

Clojure Programming

The stardog-clj source code is available as Apache 2.0 licensed code.

Installation

Stardog-clj is available from Clojars. To use, just include the following dependency:

```
[stardog-clj "2.2.2"]
```

Starting with Stardog 2.2.2, the stardog-clj version always matches the latest release of Stardog.

Overview

Stardog-clj provides a set of functions as API wrappers to the native SNARL API. These functions provide the basis for working with Stardog, starting with connection management, connection pooling, and the core parts of the API, such as executing a SPARQL query or adding and removing RDF from the Stardog database. Over time, other parts of the Stardog API will be appropriately wrapped with Clojure functions and idiomatic Clojure data structures.

Stardog-clj provides the following features:

1. Specification based descriptions for connections, and corresponding "connection" and "with-connection-pool" functions and macros
2. Functions for query, ask, graph, and update to execute `SELECT`, `ASK`, `CONSTRUCT`, and SPARQL Update queries respectively
3. Functions for insert and remove, for orchestrating the Adder and Remover APIs in SNARL
4. Macros for resource handling, including with-connection-tx, with-connnection-pool, and with-transaction
5. Support for programming Stardog applications with either the connection pool or direct handling of the connection
6. Idiomatic clojure handling of data structures, with converters that can be passed to query functions

The API with source docs can be found in the `stardog.core` and `stardog.values` namespaces.

API Overview

The API provides a natural progression of functions for interacting with Stardog

```
(create-db-spec "testdb" "snarl://localhost:5820/" "admin" "admin" "none")
```

This creates a connection space for use in `connect` or `make-datasource` with the potential parameters:

```
{:url "snarl://localhost:5820/" :db "testdb" :pass "admin" :user "admin" :max-idle
100 :max-pool 200 :min-pool 10 :reasoning false}
```

Create a single Connection using the database spec. Can be used with `with-open`, `with-transaction`, and `with-connection-tx` macros.

```
(connect db-spec)
```

Creates a data source, i.e. `ConnectionPool`, using the database spec. Best used within the `with-connection-pool` macro.

```
(make-datasource db-spec)
```

Executes the body with a transaction on each of the connections. Or establishes a connection and a transaction to execute the body within.

```
(with-transaction [connection...] body)
(with-connection-tx binding-forms body)
```

Evaluates body in the context of an active connection obtained from the connection pool.

```
(with-connection-pool [con pool] .. con, body ..)
```

Examples

Here are some examples of using stardog-clj

Create a connection and run a query

```
=> (use 'stardog.core)
=> (def c (connect {:db "testdb" :url "snarl://localhost"}))
=> (def results (query c "select ?n { .... }"))
=> (take 5 results)
({:n #<StardogURI http://example.org/math#2>} {:n #<StardogURI
http://example.org/math#3>} {:n #<StardogURI http://example.org/math#5>} {:n
#<StardogURI http://example.org/math#7>} {:n #<StardogURI
http://example.org/math#11>})

=> (def string-results (query c "select ?n { .... }" {:converter str}))
=> (take 5 string-results)
({:n "http://example.org/math#2"} {:n "http://example.org/math#3"} {:n
"http://example.org/math#5"} {:n "http://example.org/math#7"} {:n "http://example.org/
math#11"})
```

Insert data

```
(let [c (connect test-db-spec)]
               (with-transaction [c]
                 (insert! c ["urn:test" "urn:test:clj:prop2" "Hello World"])
                 (insert! c ["urn:test" "urn:test:clj:prop2" "Hello World2"]))
```

Run a query with a connection pool

```
myapp.core=> (use 'stardog.core)
nil
myapp.core=> (def db-spec (create-db-spec "testdb" "snarl://localhost:5820/" "admin"
"admin" "none"))
#'myapp.core/db-spec
myapp.core=> (def ds (make-datasource db-spec))
myapp.core=> (with-connection-pool [conn ds]
        #_=>   (query conn "SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 2"))
({:s #<URI urn:test1>, :p #<URI urn:test:predicate>, :o "hello world"} {:s #<URI
urn:test1>, :p #<URI urn:test:predicate>, :o "hello world2"})
```

SPARQL Update

```
;; First, add a triple
;; Then run an udpate query, which is its own transaction
;; Finally, confirm via ask
 (with-open [c (connect test-db-spec)]
               (with-transaction [c]
                 (insert! c ["urn:testUpdate:a1" "urn:testUpdate:b" "aloha world"]))
               (update c "DELETE { ?a ?b \"aloha world\" } INSERT { ?a ?b \"shalom
world\" } WHERE { ?a ?b \"aloha world\"  }"
                         {:parameters {"?a" "urn:testUpdate:a1" "?b"
"urn:testUpdate:b"}})
               (ask c "ask { ?s ?p \"shalom world\" }") => truthy)
```

Graph function for Construct queries

```
;; Graph results converted into Clojure data using the values methods
(with-open [c (connect test-db-spec)]
               (let [g (graph c "CONSTRUCT { <urn:test> ?p ?o } WHERE { <urn:test> ?p
?o } ")]
                 g) => (list [(as-uri "urn:test") (as-uri "urn:test:clj:prop3")
"Hello World"]))
```

Ask function for ASK queries

```
;; Ask returns a Boolean
(with-open [c (connect test-db-spec)]
               (ask c "ask { ?s <http://www.lehigh.edu/~zhp2/2004/0401/
univ-bench.owl#teacherOf> ?o }")) => truthy)
```

.Net Programming

In the Network Programming section, we looked at how to interact with Stardog over a network via HTTP and SNARL protocols. In this chapter we describe how to program Stardog from .Net using http://www.dotnetrdf.org.

NOTE

.dotNetRDF is an open source library developed and supported by third parties; questions or issues with the .Net API should be directed to http://www.dotnetrdf.org.

You should also be aware that dotNetRDF uses the HTTP API for all communication with Stardog so you **must** enable the HTTP server to use Stardog from .Net. It's enabled by default so most users should not need to do anything to fulfill this requirement.

dotNetRDF Documentation

See the documentation for using dotNetRDF with Stardog.

Spring Programming

The Spring for Stardog source code is available on Github. Binary releases are available on the Github release page.

As of 2.1.3, Stardog-Spring and Stardog-Spring-Batch can both be retrieved from Maven central:

- `com.complexible.stardog:stardog-spring:2.1.3`
- `com.complexible.stardog:stardog-spring-batch:2.1.3`

The corresponding Stardog Spring version will match the Stardog release, e.g. stardog-spring-2.2.2 for Stardog 2.2.2.

Overview

Spring for Stardog makes it possible to rapidly build Stardog-backed applications with the Spring Framework. As with many other parts of Spring, Stardog's Spring integration uses the template design pattern for abstracting standard boilerplate away from application developers.

Stardog Spring can be included via Maven with `com.complexible.stardog:stardog-spring:version` and `com.complexible.stardog:stardog-spring-batch` for Spring Batch support. Both of these dependencies require the public Stardog repository to be included in your build script, and the Stardog Spring packages installed in Maven. Embedded server is still supported, but via providing an implementatino of the Provider interface. This enables users of the embedded server to have full control over how to use the embedded server.

At the lowest level, Spring for Stardog includes

1. `DataSouce` and `DataSourceFactoryBean` for managing Stardog connections
2. `SnarlTemplate` for transaction- and connection-pool safe Stardog programming
3. `DataImporter` for easy bootstrapping of input data into Stardog

In addition to the core capabilities, Spring for Stardog also integrates with the Spring Batch framework. Spring Batch enables complex batch processing jobs to be created to accomplish tasks such as ETL or legacy data migration. The standard `ItemReader` and `ItemWriter` interfaces are implemented with a separate callback writing records using the SNARL Adder API.

Basic Spring

There are three Beans to add to a Spring application context:

- `DataSourceFactoryBean` : `com.complexible.stardog.ext.spring.DataSourceFactoryBean`
- `SnarlTemplate` : `com.complexible.stardog.ext.spring.SnarlTemplate`
- `DataImporter` : `com.complexible.stardog.ext.spring.DataImporter`

`DataSourceFactoryBean` is a Spring `FactoryBean` that configures and produces a `DataSource` . All of the Stardog `ConnectionConfiguration` and `ConnectionPoolConfig` methods are also property names of the `DataSourceFactoryBean` --for example, "to", "url", "createIfNotPresent". If you are interested in running an embedded server, use the `Provider` interface and inject it into the `DataSourceFactoryBean`. Note: all of the server jars must be added to your classpath for using the embedded server.

`javax.sql.DataSource`, that can be used to retrieve a `Connection` from the `ConnectionPool`. This additional abstraction serves as place to add Spring-specific capabilities (e.g. `spring-tx` support in the future) without directly requiring Spring in Stardog.

`SnarlTemplate` provides a template abstraction over much of Stardog's native API, SNARL, and follows the same approach of other Spring template, i.e., `JdbcTemplate`, `JmsTemplate`, and so on.

Spring for Stardog also comes with convenience mappers, for automatically mapping result set bindings into common data types. The `SimpleRowMapper` projects the `BindingSet` as a `List>` and a `SingleMapper` that accepts a constructor parameter for binding a single parameter for a single result set.

The key methods on `SnarlTemplate` include the following:

```
query(String sparqlQuery, Map args, RowMapper)
```

`query()` executes the SELECT query with provided argument list, and invokes the mapper for result rows.

```
doWithAdder(AdderCallback)
```

`doWithAdder()` is a transaction- and connection-pool safe adder call.

```
doWithGetter(String subject, String predicate, GetterCallback)
```

`doWithGetter()` is the connection pool boilerplate method for the `Getter` interface, including the programmatic filters.

```
doWithRemover(RemoverCallback)
```

`doWithRemover()` As above, the remover method that is transaction and pool safe.

```
execute(ConnectionCallback)
```

`execute()` lets you work with a connection directly; again, transaction and pool safe.

```
construct(String constructSparql, Map args, GraphMapper)
```

`construct()` executes a SPARQL CONSTRUCT query with provided argument list, and invokes the `GraphMapper` for the result set.

`DataImporter` is a new class that automates the loading of RDF files into Stardog at initialization time.

It uses the Spring Resource API, so files can be loaded anywhere that is resolvable by the Resource API: classpath, file, url, etc. It has a single load method for further run-time loading and can load a list of files at initialization time. The list assumes a uniform set of file formats, so if there are many different types of files to load with different RDF formats, there would be different `DataImporter` beans configured in Spring.

Spring Batch

In addition to the base `DataSource` and `SnarlTemplate`, Spring Batch support adds the following:

- `SnarlItemReader`: `com.complexible.stardog.ext.spring.batch.SnarlItemReader`
- `SnarlItemWriter`: `com.complexible.stardog.ext.spring.batch.SnarlItemWriter`
- `BatchAdderCallback`: `com.complexible.stardog.ext.spring.batch.BatchAdderCallback`

Groovy Programming

Groovy is an agile and dynamic programming language for the JVM, making popular programming features such as closures available to Java developers. Stardog's Groovy support makes life easier for developers who need to work with RDF, SPARQL, and OWL by way of Stardog.

The Groovy for Stardog source code is available on Github.

Binary releases are available on the Github release page and via Maven central as of version 2.1.3 and beyond using the following dependency declaration (Gradle style) `com.complexible.stardog:stardog-groovy:2.1.3`.

As of version 2.1.3, Stardog-Groovy can be included via "com.complexible.stardog:stardog-groovy:2.1.3" from Maven central.

> **NOTE** You must include our public repository in your build script to get the Stardog client dependencies into your local repository.

Using the embedded server with Stardog Groovy is not supported in 2.1.2, due to conflicts of the asm library for various third party dependencies. If you wish to use the embedded server with similar convenience APIs, please try Stardog with Spring. Also 2.1.3 and beyond of Stardog-Groovy no longer requires the use of the Spring framework.

The Stardog-Groovy version always matches the Stardog release, e.g. for Stardog 2.2.2 use stardog-groovy-2.2.2.

Overview

Groovy for Stardog provides a set of Groovy API wrappers for developers to build applications with Stardog and take advantage of native Groovy features. For example, you can create a Stardog connection pool in a single line, much like Groovy SQL support. In Groovy for Stardog, queries can be iterated over using closures and transaction safe closures can be executed over a connection.

For the first release, Groovy for Stardog includes `com.complexible.stardog.ext.groovy.Stardog` with the following methods:

1. `Stardog(map)` constructor for managing Stardog connection pools
2. `each(String, Closure)` for executing a closure over a query's results, including projecting SPARQL result variables into the closure.
3. `query(String, Closure)` for executing a closure over a query's results, passing the BindingSet to the closure
4. `insert(List)` for inserting a list of vars as a triple, or a list of list of triples for insertion
5. `remove(List)` for removing a triple from the database
6. `withConnection` for executing a closure with a transaction safe instance of `Connection`

Examples

Here are some examples of the more interesting parts of Stardog Groovy.

Create a Connection

```
def stardog = new Stardog([url: "snarl://localhost:5820/", to:"testdb",
username:"admin", password:"admin"])
stardog.query("select ?x ?y ?z WHERE { ?x ?y ?z } LIMIT 2", { println it } )
// in this case, it is a BindingSet, ie TupleQueryResult.next() called until
exhausted and closure executed
```

SPARQL Vars Projected into Groovy Closures

```
// there is also a projection of the results into the closure's binding
// if x, y, or z are not populated in the answer, then they are still valid binidng
but are null
stardog.each("select ?x ?y ?z WHERE { ?x ?y ?z } LIMIT 2", {
  println x
  println y
  println z // may be a LiteralImpl, so you get full access to manipulate Value
objects
  }
)
```

Add & Remove Triples

```
// insert and remove
stardog.insert([["urn:test3", "urn:test:predicate", "hello world"],
          ["urn:test4", "urn:test:predicate", "hello world2"]])
stardog.remove(["urn:test3", "urn:test:predicate", "hello world"])
stardog.remove(["urn:test4", "urn:test:predicate", "hello world2"])
```

withConnection Closure

```
// withConnection, tx safe
stardog.withConnection { con ->
    def queryString = """
                SELECT ?s ?p ?o
                {
                  ?s ?p ?o
                }
        """
        TupleQueryResult result = null;
        try {
                Query query = con.query(queryString);
                result = query.executeSelect();
                while (result.hasNext()) {
                        println result.next();
                }

                result.close();

        } catch (Exception e) {
                println "Caught exception ${e}"
        }
}
```

SPARQL Update Support

```
// Accepts the SPARQL Update queries
stardog.update("DELETE { ?a ?b \"hello world2\" } INSERT { ?a ?b \"aloha world2\" }
WHERE { ?a ?b \"hello world2\" }")

def list = []
stardog.query("SELECT ?x ?y ?z WHERE { ?x ?y \"aloha world2\" } LIMIT 2", { list <<
it } )
assertTrue(list.size == 1)
```

SNARL Migration Guide

Stardog 3

This section guides migration from Stardog 2.x to Stardog 3.

Reasoning

`ReasoningType` is no longer a `Connection` client option. Reasoning is now enabled, or disabled; the database will determine the appropriate OWL profile to use given the TBox associated with the database. Therefore, the CLI commands that optionally accept `-r, --reasoning` argument does not need a level value. The commands that required a reasoning level in 2.x does not need a `-r, --reasoning` argument anymore as they enable reasoning automatically.

The default value of `reasoning.schema.graphs` has been changed to `ALL` from `DEFAULT`.

Custom Functions and Aggregates

The API for custom SPARQL functions has changed in Stardog 3 to accommodate the addition of custom SPARQL aggregates. A discussion on how to create custom functions and aggregates can be found in the examples.

Database Properties

In Stardog 3, `DatabaseOptions` no longer serves as a single collection of all options for the system. Module-specific options have been split into their respective modules. For example, all options dealing with configuration of the reasoner in Stardog—those whose names are prefixed with `reasoning`, such as `SCHEMA_GRAPHS`--have been moved to `ReasoningOptions`. Similar changes were made for `ICVOptions` and `SearchOptions`.

`query.all.graphs` can now be set as a database option too. If there is no value set for this option in the database metadata, the default value specified in the stardog.properties file will be used.

The database connection timeout option is now called `database.connection.timeout` and it accepts duration strings such as `1h` or `10m`. This option can be set either in stardog.properties or in the database metadata.

Empire

The Empire bindings for Stardog have been released under Apache license and are no longer part of the Stardog distribution. They are now available in the Empire code base in the `stardog` module.

API

The following methods were added to the SNARL API:

- `Getter#ask`
- `ConnectionConfiguration#setAll`

Additionally, the Search API now exposes the ability to execute searches over named graphs, which was previously only available via SPARQL.

Stardog 2

This document guides migration of code from SNARL 1.0 to SNARL 2.0 API.

Deprecating and Renaming

- All deprecated methods have been removed.
- All `com.clarkparsia` packages have been moved to `com.complexible`.
- `com.clarkparsia.stardog.reasoning.ReasoningType` has been moved to `com.complexible.stardog.reasoning.api.ReasoningType`.
- `com.clarkparsia.openrdf.query` has been moved to `org.openrdf.queryrender`.
- Everything else in `com.clarkparsia.openrdf` has been moved to `com.complexible.common.openrdf`.
- All methods marked @Beta have been promoted.

Queries

We introduced a new hierarchy for the class `com.complexible.stardog.api.Query`:

```
+ com.complexible.stardog.api.Query
        + com.complexible.stardog.api.ReadQuery
                + com.complexible.stardog.api.BooleanQuery
                + com.complexible.stardog.api.GraphQuery
                + com.complexible.stardog.api.SelectQuery
        + com.complexible.stardog.api.UpdateQuery
```

Queries can be created from a `com.complexible.stardog.api.Connection` object using the suitable method according to desired type of query: `select`, `ask`, `graph`, or `update`.

Now you can specify the reasoning type with which a particular `com.complexible.stardog.api.ReadQuery` is to be executed via the method `reasoning(ReasoningType)`. The query reasoning type overrides the reasoning type of the parent connection. Note that setting the reasoning type to `ReasoningType.NONE` will disable reasoning for that particular query, it does not affect the default reasoning specified by the `Connection`.

The methods `executeAsk()`, `executeSelect()`, and `executeGraph()` on `com.complexible.stardog.api.Query` have been removed. Queries can be executed by using the `execute()` method which will return a value appropriate for the type of query being executed.

Connections

The class `com.complexible.stardog.api.admin.StardogDBMS` was removed. It has been replaced by `com.complexible.stardog.api.admin.AdminConnectionConfiguration` for creating connections to the Stardog DBMS and `com.complexible.stardog.api.admin.AdminConnection` for the actual connection.

The method `login` on `com.complexible.stardog.api.admin.StardogDBMS` (now `com.complexible.stardog.api.admin.AdminConnectionConfiguration`) has been renamed `connect` to align with usage of the standard `com.complexible.stardog.api.ConnectionConfiguration`

The method `connect(ReasoningType)` on `com.complexible.stardog.api.ConnectionConfiguration` has been removed.

The method `getBaseConnection()` on `com.complexible.stardog.api.reasoning.ReasoningConnection` has been removed. To obtain a `ReasoningConnection` from a base connection, simply use `Connection.as(ReasoningConnection.class)`.

Explanations

The explain functions of `com.complexible.stardog.api.reasoning.ReasoningConnection` now return `com.complexible.stardog.reasoning.Proof` objects. The `com.complexible.stardog.reasoning.Proof.getStatements()` function can be used to get only the asserted statements which is equivalent to what explain functions returned in 1.x.

Starting the server

In order to create a new server we use a `ServerBuilder` obtained via the method `buildServer()` on `com.complexible.stardog.Stardog`; configuration options can be `set(Option<T>, T)` and the server is created for a particular address with `bind`. The following example shows how to create a new embedded SNARL server.

```
Server aServer = Stardog
                            .buildServer()
                            .bind(SNARLProtocol.EMBEDDED_ADDRESS)
                            .start();
```

When programmatically starting a Stardog server in your application, you **must** stop the server when you're done with it, otherwise it can prevent the JVM from exiting.

Protocols

As of Stardog 2.0, Stardog's supported protocols, SNARL & HTTP, now run on the **same** port. There is no need to start separate servers or specify different ports. The new unified Stardog server will automatically detect what protocol you are using and forward the traffic appropriately. The default port for the server remains `5820`.

Command line

The global options `--home`, `--logfile`, `--disable-security` for `server start` command have been turned into regular options. See the `stardog-admin help server start` for details.

Understanding Stardog

Background information on performance, testing, terminology, known issues, compatibility policies, etc.

Frequently Asked Questions

Some frequently asked questions for which we have answers.

Why can't I load Dbpedia (or other RDF) data?

Question

I get a parsing error when loading Dbpedia or some other RDF. What can I do?

Answer

First, it's not a bad thing to expect data providers to publish valid data. Second, it is, apparently, a very naive thing to expect data providers to publish valid data…

Stardog supports a loose parsing mode which will ignore certain kinds of data invalidity and may allow you to load invalid data. See `strict.parsing` in Configuration Options.

Why don't my queries work?!

Question

I've got some named graphs and blah blah my queries don't work blah blah.

Answer

Queries with FROM NAMED with a named graph that is not **in** Stardog will not cause Stardog to download the data from an arbitrary HTTP URL and include it in the query. *Stardog will only evaluate queries over data that has been loaded into it*.

SPARQL queries without a context or named graph are executed against **the default, unnamed graph**. In Stardog, **the default graph is not the union of all the named graphs and the default graph**. This behavior is configurable via the `query.all.graphs` configuration parameter.

SPARQL 1.1

Question

Does Stardog support SPARQL 1.1?

Answer

Yes.

Deadlocks and Slowdowns

Question

Stardog slows down or deadlocks?! I don’t understand why, I’m just trying to send some queries and do something with the results…in a tight inner loop of doom!

Answer

Make sure you are closing result sets (`TupleQueryResult` and `GraphQueryResult` ; or the Jena equivalents) when you are done with them. These hold open resources both on the client and on the server and failing to close them when you are done will cause files, streams, lions, tigers, and bears to be held open. If you do that enough, then you’ll eventually exhaust all of the resources in their respective pools, which can cause slowness or, in some cases, deadlocks waiting for resources to be returned.

Similarly close your connections when you are done with them. Failing to close `Connections` , `Iterations` , `QueryResults` , and other **closeable** objects will lead to undesirable behavior.

Bulk Update Performance

Question

I’m adding one triple at a time, in a tight loop, to Stardog; is this the ideal strategy with respect to performance?

Question

I’m adding millions of triples to Stardog and I’m wondering if that’s the best approach?

Answer

The answer to both questions is "not really"…Generally overall update performance is best if you write between 1k and 100k triples at a time. You may need to experiment to find the sweet spot with respect to your data, database size, the size of the differential index, and update frequency.

Public Endpoint

Question

I want to use Stardog to serve a public SPARQL endpoint; is there some way I can do this without publishing user account information?

Answer

We don’t necessarily recommend this, but it’s possible. Simply pass `--disable-security` to `stardog-admin` when you start the Stardog Server. This **completely** disables security in Stardog which will let users access the SPARQL endpoint, and all other functionality, without needing authorization.

Remote Bulk Loading

Question

I’m trying to create a database and bulk load files from my machine to the server and it’s not working, the files don’t seem to load, what gives?

Answer

Stardog does not tranfser files during database creation to the server, sending big files over a network kind of defeats the purpose of blazing fast bulk loading. If you want to bulk load files from your machine to a remote server, copy them to the server and bulk load them.

Canonicalized Literals

Question

Why doesn’t my literal look the same as I when I added it to Stardog?

Answer

Stardog performs literal canonicalization by default. This can be turned off by setting `index.literals.canonical` to `false`. See Configuration Options for the details.

Cluster Isn’t Working

Question

I’ve setup Stardog Cluster, but it isn’t working and I have `NoRouteToHostException` exceptions all over my Zookeeper log.

Answer

Typically—but especially on Red Hat Linux and its variants—this means that `iptables` is blocking one, some, or all of the ports that the Cluster is trying to use. You can disable `iptables` or, better yet, configure it to unblock the ports Cluster is using.

Client Connection Isn’t Working

Question

I’m getting a `ServiceConfigurationError` saying that `SNARLDriver` could not be instantiated.

Answer

Make sure that your classpath includes all Stardog JARs and that the user executing your code has access to them.

Loading Compressed Data

Question

How can I load data from a compressed format that Stardog doesn’t support without decompressing the file?

Answer

Stardog supports several compression formats by default (zip, gzip, bzip2) so files compressed with those formats can be passed as input directly without decompression. Files compressed with other formats can also be loaded to Stardog by decompressing them on-the-fly using named pipes in Unix-like systems. The following example shows using a named pipe where the decompressed data is sent directly to Stardog without being writing to disk.

```
$ mkfifo some-data.rdf
$ xz -dc some-data.rdf.xz > some-data.rdf &
$ stardog-admin db create -n test some-data.rdf
```

Benchmark Results

Live, dynamically updated performance data from BSBM, SP2B, LUBM benchmarks against the latest Stardog release.

Compatibility Policies

The Stardog 3.x release ("Stardog" for short) is a major milestone in the development of the system. Stardog is a stable platform for the growth of projects and programs written for Stardog.

Stardog provides (and defines) several user-visible things:

1. SNARL API
2. BigPacket Message Format
3. Stardog Extended HTTP Protocol
4. a command-line interface

It is intended that programs—as well as SPARQL queries—written to Stardog APIs, protocols, and interfaces will continue to run correctly, unchanged, over the lifetime of Stardog. That is, over all releases identified by version `3.x.y`. At some indefinite point, Stardog 4.x may be released; but, until that time, Stardog programs that work today should continue to work even as future releases of Stardog occur. APIs, protocols, and interfaces may grow, acquiring new parts and features, but not in a way that breaks existing Stardog programs.

Expectations

Although we expect that nearly all Stardog programs will maintain this compatibility over time, it is impossible to guarantee that no future change will break any program. This document sets expectations for the compatibility of Stardog programs in the future. The main, foreseeable reasons for which this compatibility may be broken in the future include:

1. **Security**: We reserve the right to break compatibility if doing so is required to address a security problem in Stardog.
2. **Unspecified behavior**: Programs that depend on unspecified[41] behaviors may not work in the future if those behaviors are modified.
3. **3rd Party Specification Errors**: It may become necessary to break compatibility of Stardog programs in order to address problems in some 3rd party specification.
4. **Bugs**: It will not always be possible to fix bugs found in Stardog—or in its 3rd party dependencies—while also preserving compatibility. With that proviso, we will endeavor to only break compatibility when repairing critical bugs.

It is always possible that the performance of a Stardog program may be (adversely) affected by changes in the implementation of Stardog. No guarantee can be made about the performance of a given program between releases, except to say that our expectation is that performance will generally trend in the appropriate direction.

Data Migration & Safety

We expect that data safety will always be given greater weight than any other consideration. But since Stardog stores a user's data differently from the form in which data is input to Stardog, we may from time to time change the way it is stored such that explicit data migration will be necessary.

Stardog provides for two data migration strategies:

1. Command-line migration tool(s)
2. Dump and reload

We expect that explicit migrations may be required from time to time between different releases of Stardog. We will endeavor to minimize the need for such migrations. We will only require the "dump and reload" strategy between **major** releases of Stardog (that is, from 1.x to 2.x, etc.), unless that strategy of migration is required to repair a security or other data safety bug.

Known Issues

The known issues in Stardog 3.1.4:

1. Our implementation of `CONSTRUCT` slightly deviates from the SPARQL 1.1 specification: it does not implicitly `DISTINCT` query results; rather, it implicitly applies `REDUCED` semantics to `CONSTRUCT` query results.[42]
2. Asking for all individuals with reasoning via the query `{?s a owl:Thing}` might also retrieve some classes and properties. **WILLFIX**
3. Schema queries do not bind graph variables.
4. Dropping a database deletes all of the data files in Stardog Home associated with that database. If you want to keep the data files and remove the database from the system catalog, then you need to manually copy these files to another location before dropping the database.
5. If relative URIs exist in the data files passed to create, add, or remove commands, then they will be resolved using the constant base URI `http://api.stardog.com/` if, but only if, the format of the file allows base URIs. Turtle and RDF/XML formats allows base URIs but N-Triples format doesn't allow base URIs and relative URIs in N-Triples data will cause errors.
6. Queries with `FROM NAMED` with a named graph that is **not** in Stardog will **not** cause Stardog to download the data from an arbitrary HTTP URL and include it in the query.
7. SPARQL queries without a context or named graph are executed against the default, unnamed graph. In Stardog, the default graph is **not** the union of all the named graphs and the default graph. Note: this behavior is configurable via the `query.all.graphs` configuration parameter.
8. RDF literals are limited to 8MB (after compression) in Stardog. Input data with literals larger than 8MB (after compression) will raise an exception.

Glossary

In the Stardog documentation, the following terms have a specific technical meaning.

Stardog Database Management System, aka Stardog Server An instance of Stardog; only one Stardog Server may run per JVM. A computer may run multiple Stardog Servers by running one per multiple JVMs.

Stardog Home, aka `STARDOG_HOME` A directory in a filesystem in which Stardog stores files and other information; established either in a Stardog configuration file or by environment variable. Only one Stardog Server may run simultaneously from a `STARDOG_HOME`.

Stardog Network Home A URL (HTTP or SNARL) which identifies a Stardog Server running on the network.

Database A Stardog database is a graph of RDF data under management of a Stardog Server. It may contain zero or more RDF Named Graphs. A Stardog Server may manage more than one Database; there is no hard limit, and the practical limit is disk space.

Database Short Name, aka Database Name An identifier used to name a database, provided as input when a database is created.

Database Network Name A Database Short Name is part of the URI of a Database addressed over some network protocol.

Index The unit of persistence for a Database. We sometimes (sloppily) use Database and Index interchangeably in the manual.

Memory Database A Database may be stored in-memory or on disk; a Memory Database is read entirely into system memory but can be (optionally) persisted to disk.

Disk Database A Disk Database is only paged into system memory as needed and is persisted using one or more indexes.

Connection String An identifier (a restricted subset of legal URLs, actually) that is used to connect to a Stardog database to send queries or perform other operations.

Named Graph A Named Graph is an explicitly named unit of data within a Database. Named Graphs are queries explicitly by specifying them in SPARQL queries. There is no practical limit on the number of Named Graphs in a Database.

Default Graph The Default Graph in a Database is the context into which RDF triples are stored when a Named Graph is not explicitly specified. A SPARQL query executed by Stardog that does not contain any Named Graph statements is executed against the data in the Default Graph only.

Security Realm A Security Realm defines the users and their permissions for each Database in an Stardog Server. There is only one Security Realm per Stardog Server.

Appendix

Just move it to the Appendix for a great good!

SPARQL Query Functions

Stardog supports all of the functions in SPARQL, as well as some others from XPath and SWRL. Any of these functions can be used in queries or rules. Some functions appear in multiple namespaces, but all of the namespaces will work:

8. Table of Stardog Function Namespaces

Prefix	Namespace
stardog	tag:stardog:api:functions:
fn	http://www.w3.org/2005/xpath-functions#
math	http://www.w3.org/2005/xpath-functions/math#
swrlb	http://www.w3.org/2003/11/swrlb#
afn	http://jena.hpl.hp.com/ARQ/function#

The function names and URIs supported by Stardog are included below. Some of these functions exist in SPARQL natively, which just means they can be used without an explicit namespace.

9. Table of Stardog Function Names & URIs

Function name	Recognized URI(s)
abs	fn:numeric-abs swrlb:abs

Function name	Recognized URI(s)
acos	math:acos
add	fn:numeric-add
asin	math:asin
atan	math:atan
ceil	fn:numeric-ceil swrlb:ceiling
concat	fn:concat swrlb:stringConcat
contains	fn:contains swrlb:contains
containsIgnoreCase	swrlb:containsIgnoreCase
cos	math:cos swrlb:cos
cosh	stardog:cosh
date	swrlb:date
dateTime	swrlb:dateTime
day	fn:day-from-dateTime
dayTimeDuration	swrlb:dayTimeDuration
divide	fn:numeric-divide swrlb:divide

Function name	Recognized URI(s)
encode_for_uri	fn:encode_for_uri
exp	math:exp afn:e
floor	fn:numeric-floor swrlb:floor
hours	fn:hours-from-dateTime
integerDivide	fn:numeric-integer-divide swlrb:integerDivide
lcase	fn:lower-case swlrb:lowerCase
log	math:log
log10	math:log10
max	fn:max afn:max
min	fn:min afn:min
minutes	fn:minutes-from-dateTime
mod	swrlb:mod
month	fn:month-from-dateTime
multiply	fn:numeric-multiply

Function name	Recognized URI(s)
normalizeSpace	fn:normalize-space swrlb:normalizeSpace
pi	math:pi afn:pi
pow	math:pow swrlb:pow
replace	fn:replace
round	fn:numeric-round swrlb:round
roundHalfToEven	fn:numeric-round-half-to-even swrlb:roundHalfToEven
seconds	fn:seconds-from-dateTime
sin	math:sin swrlb:sin
sinh	stardog:sinh
sqrt	math:sqrt afn:sqrt
strafter	fn:substring-after swrlb:substringAfter
strbefore	fn:substring-before swrlb:substringBefore

Function name	Recognized URI(s)
strends	fn:ends-with swrlb:endsWith
stringEqualIgnoreCase	swrlb:stringEqualIgnoreCase
strlen	fn:string-length swrlb:stringLength
strstarts	fn:starts-with swrlb:startsWith
substring	fn:substring swrlb:substring
subtract	fn:numeric-subtract swrlb:subtract
tan	math:tan swrlb:tan
tanh	stardog:tanh
time	swrlb:time
timezone	fn:timezone-from-dateTime
toDegrees	stardog:toDegrees
toRadians	stardog:toRadians
translate	fn:translate swrlb:translate
ucase	fn:upper-case swrlb:upperCase

Function name	Recognized URI(s)
`unaryMinus`	`fn:numeric-unary-minus` `swrlb:unaryMinus`
`unaryPlus`	`fn:numeric-unary-plus` `swrlb:unaryPlus`
`year`	`fn:year-from-dateTime`
`yearMonthDuration`	`swrlb:yearMonthDuration`

Milestones

This timeline describes major features and other notable changes to Stardog starting at 1.0; it will be updated for each notable new release. For a complete list of changes, including notable bug fixes, see the release notes.

3.1

- Named Graph security
- BOSH-based cluster management tool
- proper logshipping in the Cluster

3.0

- Equality reasoning via hybrid materialization
- Improved incremental write performance
- HA Cluster production ready
- Integrity constraint violation repair plans
- Improved query performance

2.2.1

- Stardog HA Cluster (beta)

2.2

- Support for RDF versioning
- Admin support for Web Console

2.1

- Database repair, backup & restore utilities
- Improved query scalability by flowing intermediate results off-heap or onto disk; requires a JDK that supports `sun.misc.Unsafe`
- Performance: significant improvement in performance of bulk loading and total scalability of a database
- Generation of multiple proofs for inferences & inconsistencies; proofs for integrity constraint violations
- Reduced memory footprint of queries while being executed

2.0

- SPARQL 1.1 Update: the most requested feature ever!
- Stardog Web Console: a Stardog Web app for managing Stardog Databases; includes Linked Data Server, etc.
- JMX monitoring: includes graphical monitoring via Web Console
- HTTP & SNARL servers unified into a single server (default port 5820)
- Database Archetypes for PROV, SKOS; extensible for user-defined ontologies, schemas, etc.
- Stardog Rules Syntax: new syntax for user-defined rules
- Performance improvements for SPARQL query evaluation
- Hierarchical explanations of inferences using proof trees
- SL reasoning profile
- Client and server dependencies cleanly separated
- Evaluation of non-recursive datalog queries to improve reasoning performance

1.2

- Query management: slow query log, kill-able queries, etc.
- new CLI
- new transaction layer
- SPARQL Service Description
- new security layer
- Query rewrite cache
- Removed Stardog Shell

1.1.2

- New optimizer for subqueries

1.1

- SPARQL 1.1 Query
- Transitive reasoning
- User-defined rules in SWRL
- new SWRL builtins and syntactic sugar for schema-queries
- Improved performance of reasoning queries involving rdf:type
- Improved performance of search indexing
- Deprecated Stardog Shell

1.0.4

- Convert ICVs to SPARQL queries in the CLI or Java API
- Running as a Windows Service
- Parametric queries in CLI

1.0.2

- Stardog Community edition introduced
- ICV in SNARL and HTTP

- HTTP Admin protocol extensions
- SPARQL 1.1 Graph Store Protocol

1.0.1

- Self-hosting Stardog documentation
- Prefix mappings per database
- Access and audit logging

1.0

- Execute `DESCRIBE` queries against multiple resources
- Database consistency checking from CLI
- Inference explanations from CLI

[1] Robert Butler, Marko A. Rodriguez, Brian Sletten, Alin Dreghiciu, Rob Vesse, Stephane Fallah, John Goodwin, José Devezas, Chris Halaschek-Wiener, Gavin Carothers, Brian Panulla, Ryan Kohl, Morton Swimmer, Quentin Reul, Paul Dlug, James Leigh, Alex Tucker, Ron Zettlemoyer, Jim Rhyne, Andrea Westerinen, Huy Phan, Zach Whitley, Maurice Rabb, Grant Pax, Conrad Leonard, the crew at XSB, and the nice people at El Rinconcito Cafe in DC.

[2] That is, the server listens to one port (5820) and handles both protocols.

[3] That is, the server listens to one port (5820) and handles both protocols.

[4] The last SPARQL 1.1 feature that we didn't support.

[5] User feedback about this limitation is welcomed.

[6] Find another database that can do that!

[7] In other words, if there is a conflict between this documentation and the output of the CLI tools' `help` command, the CLI output is correct.

[8] It's very unlikely you'll need this.

[9] We're big fans of /opt/stardog/{$version} and setting `STARDOG_HOME` to /var/stardog` but YMMV.

[10] This is equally true, when using Stardog HA Cluster, of Zookeeper's access to free disk space. Very bad things happen to the Stardog Cluster if Zookeeper cannot write to disk.

[11] For more details about configuring these values, see the `examples/stardog.properties` file that is distributed with Stardog.

[12] Compressed data may only be loaded at database creation time. We will support adding compressed data to an existing database in a future release.

[13] However, there may be some delay since Stardog only periodically checks the `query.timeout` value against internal query evaluation timers.

[14] A good general purpose discussion of these issues in context of J2EE is this beginner's guide.

[15] As discussed in SPARQL Update, since Update queries are implicitly atomic transactional operations, which means you shouldn't issue an Update query within an open transaction.

[16] However, `READ COMMITTED` does allow for non-repeatable reads.

[17] The probability of recovering from a catastrophic transaction failure is inversely proportional to the number of subsequent write attempts; hence, Stardog offlines the database to prevent subsequent write attempts and to increase the likelihood of recovery.

[18] Stardog also uses file handles and sockets, but we don't discuss those here.

[19] These are conservative values and are somewhat dataset specific. Your data may require less memory..or more!

[20] In other words, embedded Stardog access is inherently **insecure** and should be used accordingly.

[21] The Stardog HTTP client driver uses an `X509TrustManager`. The details of how a trust store is selected to initialize the trust manager are http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#X509TrustManager.

[22] See the `javax.net.ssl.trustStorePassword` system property docs: http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#X509TrustManager.

[23] The matching algorithm used is described—http://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html-- in the Apache docs about `BrowserCompatHostnameVerifier`.

[24] "Client" here means the client of Stardog APIs.

[25] "Because Zookeeper requires a majority, it is best to use an odd number of machines. For example, with four machines ZooKeeper can only handle the failure of a single machine; if two machines fail, the remaining two machines do not constitute a majority. However, with five machines ZooKeeper can handle the failure of two machines." See Zk Admin for more.

[26] Based on customer feedback we may relax these consistency guarantees in some future release. Please get in touch if you think an eventually consistent approach is more appropriate for your use of Stardog.

[27] In order to deploy a cluster to a single machine—for example, for testing or development—you must ensure that each Stardog and ZooKeeper servers has a different home location.

[28] In 3.x we will integrate Netflix's Exhibitor for Stardog Cluster management. Ping us if this is a priority for you.

[29] You only pay for the reasoning that you use; no more and no less. Eager materialization is mostly a great strategy for hard disk manufacturers.

[30] Sometimes called a "TBox".

[31] Find another database, any other database anywhere, that can do that! We'll wait...

[32] Triggered using the `--format tree` option of the `reasoning explain` CLI command.

[33] Quick refresher: the `IF` clause defines the conditions to match in the data; if they match, then the contents of the `THEN` clause "fire", that is, they are inferred and, thus, available for other queries, rules, or axioms, etc.

[34] Of course if you've tweaked `reasoning.schema.graphs`, then you should put the rules into the named graph(s) that are specifed in that configuration parameter.

[35] This is effectively the only setting for Stardog prior to 3.0.

[36] These are harmless and won't otherwise affect query evaluation; they can also be added to the data, instead of to queries, if that fits your use case better.

[37] The standard inference semantics of OWL 2 do not adopt the unique name assumption because, in information integration scenarios, things often have more than one name but that doesn't mean they are different things. For example, when several databases or other data sources all contain some partial

information about, say, an employee, but they each name or identify the employee in different ways. OWL 2 won’t assume these are different employees just because there are several names.

[38] Strictly speaking, this is a bit misleading. Stardog ICV uses both open and closed world semantics: since inferences can violate or satisfy constraints, and Stardog uses open world semantics to calculate inferences, then the ICV process is compatible with open world reasoning, to which it then applies a form of closed world validation, as described in this chapter.

[39] This is a good example of open world and closed world reasoning interacting for the win.

[40] You won’t be careful enough.

[41] The relevant specs include the Stardog-specific specifications documented on this site, but also W3C (and other) specifications of various languages, including SPARQL, RDF, RDFS, OWL 2, HTTP, Google Protocol Buffers, as well as others.

[42] Strictly speaking, this is a Sesame parser deviation from the SPARQL 1.1 spec with which we happen to agree.

Version 3.1.4

? For comments, questions, or to report problems with this page, visit the Stardog Support Forum.

COMPLEXIBLE