

Guide

Introduction

- Versions
- Main features
- Organizations
- Licenses
- Social
- Citations
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

FAQ

Meta

- Glossary
- Release notes

Introduction

Ontop is a Virtual Knowledge Graph system. It exposes the content of arbitrary relational databases as knowledge graphs. These graphs are virtual, which means that data remains in the data sources instead of being moved to another database.

Ontop translates [SPARQL queries](#) expressed over the knowledge graphs into SQL queries executed by the relational data sources. It relies on [R2RML mappings](#) and can take advantage of lightweight ontologies.

Versions

This documentation is for Ontop 3.0 and more recent versions.

Most recent version:

- Stable: Ontop 5.5.0, released on February 14, 2026.

See [release notes](#) for more details.

Main features

- Uses [RDF 1.1](#) as graph data model
- Supports RDFS and OWL 2 QL ontologies
- Supports [R2RML](#) and Ontop mappings
- Supports [the majority of SPARQL 1.1 features](#), including the main SPARQL aggregation functions (since 4.0.0) and [GeoSPARQL functions](#) (since 4.1.0)
- Can be deployed as [a SPARQL endpoint](#) and as [a predefined query endpoint](#) (since 4.1.0)
- Produces efficient SQL queries by applying many optimizations
- Supports the main database systems: [PostgreSQL](#), [MySQL](#), [MariaDB](#) (since 5.0.0), [SQL Server](#), [Oracle](#), [DB2](#), [Snowflake](#)

- (since 5.0.0), [Databricks](#) (since 5.0.0), [Google BigQuery](#) (since 5.0.2), [AWS Redshift](#) (since 5.0.2), [DuckDB](#) (since 5.0.2), [AWS DynamoDB](#) (since 5.1.0), and [TDengine](#) (since 5.4.0)
- Supports database federators such as [Denodo](#), [Dremio](#) (since 4.1.0), [Teiid](#) (since 4.1.1), [Apache Spark](#) (since 4.2.0) and [Trino / PrestoDB / AWS Athena](#) (since 5.0.2)
- Supports [lenses](#) which are "virtual views" specified outside of the data sources (since 4.2.0)
- Can [materialize](#) virtual graphs into RDF files
- Provides a plugin for editing and testing mappings in the [Protégé ontology editor](#)

Organizations

Ontop is backed by the [Free University of Bozen-Bolzano](#) and [Ontopic s.r.l.](#) . It also receives regular important contributions from [Birkbeck, University of London](#) and [Southeast University](#) . See the [community section](#) for more details.

Licenses

Ontop is available under the [Apache 2.0](#) license.

All the documentation is licensed under the [Creative Commons \(Attribution\)](#) license.

Social

You can find us on the following social platforms:

- [Twitter \(ontop4obda\)](#)
- [Mastodon \(@ontop@fosstodon.org\)](#)
- [Facebook \(obdaontop\)](#)
- [Google Group \(ontop4obda\)](#)
- [GitHub \(ontop/ontop\)](#)

Citations

- If you use Ontop in your work, please cite one of the following articles describing the system.
 - Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalayci, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. [The Virtual Knowledge Graph System Ontop](#) . In: International Semantic Web Conference (Resource Track), 2020.
 - Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. [Ontop: Answering SPARQL Queries over Relational Databases](#) . In: Semantic Web Journal 8.3 (2017), pp. 471–487.
- If you want to cite the techniques behind Ontop, check [our publications](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[Key concepts](#) →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary

Release notes

- 5.5.0 (February 14, 2026)
- 5.4.0 (September 29, 2025)
- 5.3.1 (September 29, 2025)
- 5.3.0 (February 18, 2025)
- 5.2.1 (November 24, 2024)
- 5.2.0 (August 16, 2024)
- 5.1.2 (January 17, 2024)
- 5.1.1 (November 22, 2023)
- 5.1.0 (August 15, 2023)
- 5.0.2 (March 9, 2023)
- 5.0.1 (January 29, 2023)

Release notes

5.5.0 (February 14, 2026)

New features

- Support paths to files/directories as relation names in FROM clauses in mappings [#905](#)
- Common Table Expression in SQL serialization [#911](#)
- Basic operations on constant time intervals [#916](#)
- New SPARQL function: `obdaf:queryId` [#924](#)

Refactoring

- Refactoring IQTree creation and transformation [#906](#)
- SQLQueryParser simplifications [#907](#)
- DB metadata extraction adapted for DuckDB 1.4 [#915](#)

[Milestone on Github](#)

5.4.0 (September 29, 2025)

New features

- New query unfoldr, performing query unfolding in two phases for handling abstract SPARQL queries more efficiently, added [#846](#)
- Support for `rdfs:subClassOf*` property path added [#832](#)
- Bottom-up implementation of `EXISTS`, supporting most cases for `EXISTS`, added [#853](#) and [#861](#)
- Support for TDEngine added [#875](#) and [#877](#)
- Support for multi-line target queries in OBDA files added [#874](#)

New optimization

- Case-insensitive substring matching simplified [#882](#)

Refactoring

- Robust to primary keys with NULLable attributes added [#859](#)
- `IQTree` and `QueryNode` code cleaned up [#862](#)
- Numeric binary operators refactored [#893](#)
- More Trino functions recognized [#849](#)

Bug fixes

- `ORDER BY` lifted above `DISTINCT` when necessary [#847](#)
- Foreign key inference in lenses preserved in the presence of IRI safeness declarations [#880](#)
- Correct datatypes declared for all geospatial related functions

[Milestone on Github](#)

5.3.1 (September 29, 2025)

New features

- Database init script added [#856](#)
- `ST_Distance` function for Trino added [#845](#)
- H2 information schema tables are now classified as system tables [#869](#)

New optimization

- Conjunctive query containment check now handles NULLable columns [#872](#)

Refactoring

- `CHARACTER VARYING` and `VARCHAR` are now treated the same in Dremio [#881](#)

Bugfixes

- Arity violation fixed [#866](#)
- DuckDB catalogs are now taken into account [#867](#)
- Non-renamed variable in `NullableFDSelfLJOptimizer` fixed

#900

- Missing padding with nulls when simplified left joins fixed
- Long artificial `LIMIT` value reduced for supporting Dremio 26

[Milestone on Github](#)

5.3.0 (February 18, 2025)

Major dependency update

- RDF4J updated from 4.x to 5.1.0 #819
- OWLAPI updated to 5.5.1 #819

New optimizations

- Unique constraint inference for unions improved in presence of disjoint IRI templates
- Inference of strict dependents as a lightweight alternative to functional dependencies added

Bugfix

- `STRSTARTS` handling of IRI template strings fixed

5.2.1 (November 24, 2024)

- Column info extraction for Oracle made robust to latency #827
- Implement RDF4J `Query.getDataset()` and `Query.setDataset()`
- Exponentially slow removal of variables of a large BGP after applying a distinct fixed

Apache Commons IO dependency updated

5.2.0 (August 16, 2024)

New features

- Extract user information (ID, group, role) from HTTP headers and make it available to special SQL functions #753

- Insert user information into the query log [#768](#)
- Compression option added to the materialization [#797](#)

Optimizations

- Left join optimizations:
 - Pruning the right child when unused and cardinality doesn't matter [#752](#)
 - More non-nullability constraints extracted from the ancestors for transferring self-left-joins based on functional dependencies [#761](#)
 - Self-left-join elimination with nullable determinants of functional dependencies [#783](#)
 - Self-left join elimination on nullable functional dependency determinants made robust to provenance variables and trivial functional dependencies [#804](#)
- Unique constraint and variable non requirement inference with distinct nodes improved [#756](#)
- Simplification of conjunctions and disjunctions of equalities [#755](#)
- Propagation of foreign keys from basic lenses to parents [#759](#)
- Functional dependency inference for unions improved [#760](#)
- Optimization to the SPARQL `strStartsWith` function [#801](#)
- IRI decomposition with more than 1 argument when some arguments are nullable enabled [#809](#)
- Removal of distincts below joins when there is a `limit 1` above
- Year extraction from timestamps with DuckDB optimized

Refactoring

- T-mapping and related optimizations refactored [#769](#)
- B-node label anonymization pushed into the IQTree [#772](#)
- Handling environment variables with the CLI refactored [#814](#)
- Robust simplification for GeoSPARQL functions [#808](#)
- Wrapping a values node into a lens at mapping processing time added (disabled by default; see property `ontop.wrapMappingValuesNodesInLenses`)
- Non-nullable variables are not used anymore for right provenance from a left join (was preventing some optimizations)

Bug fixes

- `<expression> = true` issue with SQLServer and Oracle fixed [#758](#)
- Connection test for Trino and H2 fixed [#767](#)
- Mysql datetime conversion is now encoded to UTF8 to avoid collation error [#779](#)
- Missing spaces in string format pattern with MySQL added [#784](#)
- Configuration classes now use injector suppliers [#794](#)
- MS SQL Server identifiers are now treated not case-sensitive (even if quoted) [#805](#)
- Incorrect time zone format in xsd:dateTime for Redshift's `TIMESTAMP` data type fixed [#807](#)
- Database constraints were ignored by Ontop CLI [#777](#)
- Error with the CLI during the R2RML to OBDA mapping conversion fixed [#799](#)
- `TIMESTAMP WITH TIME ZONE` is now recognized for H2
- Typing nulls for `DATE` with Dremio fixed
- Missing minutes from the DuckDB time zone added
- Invalid URNs for the triple patterns generated when converting the mapping to R2RML fixed
- Error with constant in order-by involving a VALUES node fixed
- Missed duplicate elimination for some overlapping IRI templates fixed
- Incorrect content types returned for CSV and TSV results of a SPARQL query fixed

[Milestone on Github](#)

5.1.2 (January 17, 2024)

Bug fixes

- Conflict between columns in case of a self-join in the bootstrapper [#786](#)
- Handling of casts for different versions of MySQL and MariaDB [#787](#)

Several dependencies updated, including JQuery ([#780](#))

5.1.1 (November 22, 2023)

New features

- Option to expose system tables added [#744](#)
- Standard deviation and variance functions added [#735](#)
- Option to disable the registration of custom SPARQL aggregation functions [#746](#)

Bug fixes

See <https://github.com/ontop/ontop/milestone/23?closed=1>

5.1.0 (August 15, 2023)

New features

- Support for [flatten lenses](#) added (*beta*) [#622](#) , [#625](#) , [#628](#) , [#629](#) , [#633](#) , [#635](#) , [#636](#) , [#637](#) , [#638](#) , [#639](#) , [#640](#) , [#641](#) , [#642](#) , [#643](#) , [#661](#) , [#715](#)
- Support for [union lenses](#) added (*beta*) [#612](#)
- Support for XSD cast functions added [#630](#)
- Support for date-trunc and more date extraction functions added [#705](#)
- Support for a separate RDF file for facts added [#666](#)
- Options to ignore invalid mapping entries [#631](#) and lenses [#709](#) added
- Partial support for regular expressions in SQL Server added [#702](#)
- Support for DynamoDB using the CData connector added (*beta*) [#693](#)

Optimizations

- Upwards propagation of lens unique constraints added [#668](#)
- Left join optimizations:
 - Reducing left joins to inner joins when the right child has a left join [#662](#)
 - Merging left joins nested on the left [#663](#) , [#686](#)
 - Robustness to nullable joining columns [#667](#)
 - Pruning the right of a left join when not contributing

- VALUES node decomposition added [#680](#)
- Functional dependency inference added [#681](#)
- Reduction of aggregation nodes with no aggregation functional terms to distinct added
- Transitive closure of functional dependencies added [#732](#)

Refactoring

- Substitutions refactored [#616](#)
- Typing for math operators added [#706](#)
- QuotedIDFactory refactored [#644](#)
- Removal of non-required variables improved [#669](#)
- Support for the STRING datatype for DuckDB added [#646](#)
- Support for SAMPLE at the database level added [#660](#)

Bug fixes

- Support LIMIT/OFFSET in SQL Server improved [#676](#) , [#679](#) , [#682](#) , [#721](#)
- Extraction of unique constraints in Oracle made more consistent [#695](#)
- Metadata extraction for DuckDB made more robust [#687](#)
- Missing casts when using Bnode templates added [#657](#)
- Invalid SPARQL UNION translation fixed [#651](#)
- Empty results for CONSTRUCT queries with unions and different variables fixed [#652](#)

Documentation

- Dedicated pages for each supported database (for instance [AWS Athena](#))
- [Tutorial about lenses](#)
- [Tutorial about materialization](#)

5.0.2 (March 9, 2023)

New features

- Support for Trino [#599](#) , PrestoDB [#601](#) , AWS Athena [#602](#) , DuckDB [#604](#) , AWS Redshift [#605](#) and Google BigQuery [#615](#)

added

- Declare some columns as IRI-safe in lenses [#587](#)
- Support for arbitrary JDBC properties added [#606](#)

Bugfixes

- Missing TLS cypher suite JRE modules added to the Docker image
- More system schemas are now ignored in Oracle [#591](#)

Tooling

- [Scaffolding tool](#) created to make it easier to implement a new SQL dialect adapter.

5.0.1 (January 29, 2023)

New feature

- DB metadata extraction for MS Access added [#589](#)
- JRE-embedded Ontop-Protégé bundles with Protégé-5.6.0-beta-2 are now provided

Refactoring

- Support for Spark SQL improved and tests added in the CI
- All the tables (besides excluded ones) in Oracle are now extracted, not just the user's ones

Several dependencies updated.

5.0.0 (December 31, 2022)

Important changes

- Java 11 or newer is required
- Ontop's Protégé plugin now requires at least Protégé 5.6-beta
- RDF4J bindings updated to 4.1.0
- The module `ontop-owlapi` is not published anymore. We recommend using `ontop-rdf4j` instead
- RDF4J workbench modules removed

- Docker image moved to `ontop/ontop`
- Ontop views renamed [lenses](#)

New features

- Support for SPARQL rules for the extending the VKG added [#576](#)
- Support for SPARQL `FROM` and `FROM NAMED` constructs (also part of [#576](#))
- The [Docker image](#) now also supported the ARM64 architecture and allows to run all the CLI commands [#532](#)
- Support for Snowflake added [#520](#)
- Support for MariaDB added [#271](#)
- N-Triples and N-quads are now supported by the SPARQL endpoint [#566](#)
- Option to fully reformulate the SPARQL query into SQL added [#577](#)
- Experimental support for flatten lenses added (only for PostgreSQL)

New optimizations

- Optimization of joins with a union of class definitions added [#543](#)
- Optimizations for generic queries with limit added [#578](#)
- Optimization for queries retrieving all the properties or classes in use in the VKG added [#581](#)

Refactoring

- New module for testing dialects in the Github CI pipeline added [#568](#)
- JSQParser updated to 4.4
- Ontop's Protégé plugin isolated from Ontop modules [#278](#)
- Support for Databricks (Apache Spark) improved

Many bugfixes. See also [its milestone on Github](#).

4.2.2 (November 18, 2022)

New features

Support for Protege 5.6-beta under Java 11 and Java 17

- Support for the Databricks-specific JDBC driver [#554](#)
- Support for LIMIT/OFFSET in SQL Server 2008 [#531](#)
- Support for materialized views in PostgreSQL [#541](#)

Bugfixes

- VALUES handling with Teiid [#525](#)
- Metadata extraction for Dremio of schema names with dots
- Typo in registration of Oracle factories [#557](#)

Preventive update of dependencies including Jackson and Gson to address potential vulnerabilities.

4.2.1 (April 20, 2022)

New feature

- Support for partial serialized DB metadata, when `ontop.allowRetrievingBlackBoxViewMetadataFromDB` is enabled.

Bugfixes

- Not renamed variable in sub-query with a distinct ([#417](#)).
- OBDA to R2RML conversion: require mapping entry IDs to be unique.
- Lang tags for constants in R2RML were ignored.
- Several fixes related to VALUES.
- Better HTTP codes for the ontology download endpoint.

The Jackson and Spring Boot dependencies have been preventively updated (to 2.13.2.2 and 2.6.6 respectively) to address potential vulnerabilities ([CVE-2020-36518](#) and [CVE-2022-22965](#)).

4.2.0 (December 30, 2021)

New features

- Support for Apache Spark ([#422](#)).

Support for time functions added ([#478](#)).

- Support for the `IN` SPARQL function added.
- Datatypes can be extracted from non-supported source queries in the mapping (treated internally as "black-box" views). Disabled by default (see `ontop.allowRetrievingBlackBoxViewMetadataFromDB`).
- TBox triples (e.g. sub-classes, domains and ranges) can be added to the default RDF graph. Disabled by default (see `ontop.enableFactExtractionWithTBox`).
- Support for [Ontop join views](#) added.
- Support for arbitrary levels of Ontop views added (views over views).
- Better integrity constraint extraction for Ontop basic and join views.
- Non-null information can now be specified for Ontop views.
- Support for b-node facts added.
- The default query of the SPARQL endpoint portal is now configurable ([#454](#)).
- Contextually injective IRI templates with numeric or UUID arguments are now decomposed ([#363](#)).
- JSON-LD serialization now supported by the materializer ([#481](#)).

Refactoring

- Support for Java ≥ 8 . Ontop decoupled from Protégé so as to update several common libraries ([#467](#)).
- RDF facts provided by the ontology file are now internally grouped together into values blocks (sorts of "in-memory tables") instead of being splitted into large unions. Enabled by default (see `ontop.enableValuesNode`).
- Unions are lifted above aggregations when possible.
- Docker image updated to JRE 11.
- Mutable intermediate queries are not used anymore.
- Stronger handling of non well-designed `OPTIONAL` blocks.
- Stronger handling of `ORDER BY` in sub-queries.
- Type extractor relaxed ([#438](#)) to tolerate slightly different datatypes.
- JDBC distinct result set removed.
- JDBC user name and password are not required anymore. Useful when passed in the JDBC URL.

- The Command-Line Interface of the bootstrapper has been refactored (invalid options removed, comments improved).

Many bugfixes (see for instance [our Github bugtracker](#)).

4.1.1 (July 23, 2021)

Bugfixes

- Bugs related to GeoSPARQL support ([#428](#) ,[#434](#))
- Issue with ORDER BY and LIMIT for MS SQL ([#225](#) ,[#412](#))

New features

- Teiid support ([#336](#))
- Various improvements of the Protégé plugin

4.1.0 (February 28, 2021)

Changed behavior

- [OBDA to R2RML mapping conversion](#) now requires by default access to DB metadata (through credentials or as a serialized file). This requirement can be bypassed with the argument `--force` .
- By default, DESCRIBE queries only return the triples where the described IRI appears as subject. To also consider the triples where it appears as object, set the parameter `ontop.includeFixedObjectPositionInDescribe` to true.
- The Docker endpoint image is now based on Debian ([#394](#)), not on Alpine anymore.

New features

- Support for GeoSPARQL functions added ([#335](#)).
- [Rich JSON-based query logging](#) added.
- [New endpoint for predefined queries](#) added (beta). It enables the specification of custom low-latency Web APIs in a declarative manner.
- Support for the `IF` , `IRI` , `BNODE` SPARQL functions added.
- Support for Dremio added.

- DB metadata can now be loaded from files instead of connecting to the database.
- Bnode labels are now anonymized on-the-fly. Bnode templates can safely use PII.
- Sensitive JDBC information (user name, password, URL) can now be passed as arguments or environment variables (for Docker) instead of being written in the properties file. Docker secrets are also supported for further security.
- New left join optimization techniques added. General functional dependencies are now taken into account. Sensitivity to left join ordering reduced.
- [HTTP caching](#) headers can now be returned by the Ontop SPARQL endpoint.
- Native Protégé bundles for each platform (Windows, MacOS, Linux) with JRE 8 embedded.
- Basic support for Ontop views added (experimental), which, in particular, allows specifying integrity constraints on views defined at the Ontop level.

Removal

- As announced earlier, the RDF4J Workbench-based bundles are not shipped anymore. However, the webapps war file can still be built with Maven.

Refactoring

- Drastic reduction of memory consumption when processing SPARQL queries ([#370](#)), which is significant for materializing large RDF graphs.
- Better file resource handling ([#368](#)).
- Distinct lifting improved.
- Self-join elimination based on functional dependencies re-implemented.
- Decomposition of heterogeneous IRI templates enabled.
- Lexical value space is now considered when decomposing IRIs.
- Protégé plugin codebase cleaned, with parts of UI streamlined and redesigned. Improved syntax highlighting in target and source queries. Mapping validation extended. Multiple open ontologies supported ([#282](#) and [#175](#)).

- RDF4J external binding management refactored.
- Bnode handling refactored.
- Java 11 support.
- Better handling of column names when converting OBDA mappings into R2RML, which now uses DB metadata.
- CONSTRUCT and DESCRIBE query handling refactored.
- Parser for OBDA mappings refactored.
- Character encoding in R2RML templates refactored.
- Default file encoding set to UTF-8. Particularly useful on Windows. Can be overridden if needed.
- [Build script](#) refactored. Now fully based on Maven ([#393](#)).
- Base Docker image updated.
- PostgreSQL transaction handling (in streaming mode) improved.
- Timeout support extended to non-SELECT queries.

Many bugfixes.

4.0.3 (October 16, 2020)

Bugfixes

- Wrong IRI decomposition ([#358](#))
- Logback config file ignored by Ontop CLI ([#361](#))

4.0.2 (September 3, 2020)

- Fixed Named Graphs (quad) grammar in the target of the Ontop native mapping language.

4.0.1 (August 28, 2020)

New features and bugfixes

- Named Graphs (quad) are supported in materialization ([#343](#))
- Removal of Jena as a dependency ([#349](#))
- `com.mysql.cj.jdbc.Driver` is now also recognized. Non-recognized drivers now use the default SQL factories instead of the mock-up one.

4.0.0 (July 31, 2020)

Refactoring

- Dependencies updated (e.g. RDF4J 3.3.0)
- SERIAL datatypes of PostgreSQL better handled
- Anonymous blank nodes are now supported in CONSTRUCT blocks
- JSON-LD now supported for graph queries
- Denodo support improved.

Few bugfixes.

4.0.0-rc-1 (July 8, 2020)

Changed behavior

- Streaming mode is enabled by default for query answering. It fetches data by batches of 500 results.

New features

- Support of GROUP_CONCAT added. All the standard SPARQL aggregation functions are now supported.
- Named graphs are now supported.
- Support of Denodo added. Support of Dremio is postponed to 4.1.
- Support XML catalog file added (beta). It enables resolving `owl:imports` assertions from the ontology by loading local files instead of fetching remote URLs. It is available in Protégé and with the `ontop endpoint` CLI command.
- DB metadata extraction CLI command added (experimental).
- CASE and CAST is now supported in the mapping.
- Streaming mode for query answering added.

Deprecation

- The RDF4J Workbench-based SPARQL endpoint is deprecated. Tomcat, Jetty bundles and webapps (.war files) will be removed in a future version. Users are recommended to switch to the `ontop`

`endpoint` CLI or its Docker image.

Removal

- The Mapping Assistant panel has been removed from the Ontop Protégé plugin.

Refactoring

- SQL parsing of the mapping improved. JSQLParser's version has been updated.
- Metadata extraction refactored, and SQLDialectAdapter has been eliminated.
- Meta-mapping management improved.
- Non-recognized SQL functions in the mapping are now only assumed to be deterministic. Other assumptions have been removed.
- Internal data nodes now create variables only for database columns that are used in the query. This sparse usage of variables strongly improve reformulation time and log readability when using tables with 100+ columns in the mapping.

Many bugfixes.

4.0.0-beta-1 (December 23, 2019)

Changed behaviors

- Cardinality is now enforced by default (duplicates are removed at the mapping level). This is required to support most of the aggregation functions.
- Functions in the SQL queries of the mapping that are unknown to Ontop do no cause the creation of SQL views anymore. The strategy regarding SQL functions moved from white-list to black-list.
- Types returned by supported SPARQL functions now strictly follow the SPARQL 1.1 specification.

New features

SPARQL Aggregation functions SUM, COUNT, AVG, MIN, MAX are now supported. SAMPLE is currently implemented as MIN.

- Joins over terms that are constructed using different functions are now supported.
- Some functions (such as IRI templates) can now be post-processed (that is, they are not always included in the SQL query anymore). Generated SQL queries are generally less verbose than before.
- The Ontop endpoint now has the portal option, where predefined queries are shown in the Web interface.
- The NULLIF SQL function is now supported and optimized. Useful for filtering out some unwanted values in the mapping assertions.
- OPTIONALs outside the well-designed fragment are now supported. Weakly well-designed queries encoding preferences are optimized.

Refactoring

- The typing system have been entirely redesigned. RDF types and database types are now clearly separated. Each database engine can now bring in its own data types.
- The Datalog data structure (from version 1.x) for representing queries internally has been removed.
- Function symbol creation and optimization has been entirely refactored. They are constructed by two factories, the FunctionSymbolFactory and the DBFunctionSymbolFactory.
- Most of the logic of SQLDialectAdapter has been moved to DBFunctionSymbolFactory and DBTypeFactory. Each database engine is expected to have its own implementation of these new interfaces.
- The SPARQL translation has been rewritten (was formerly translating into Datalog).
- The SQL generator has been rewritten (was formerly translating from Datalog).

Temporary limitation

- Due to a complete refactoring of the SQL dialect management, only the H2, MySQL, PostgreSQL, Oracle, SQL server and DB2 dialects are supported in this beta. Dremio and Denodo dialects

are expected to be supported in a next beta release.

Many bugfixes.

3.0.1 (December 9, 2019)

Changed behaviors

- Protégé: no property file (resp. mapping file) is created if no data source (resp. mapping assertion) is specified by the user.

New features

- Ontop endpoint now supports R2RML files and doesn't require an ontology.
- Materialization: N-triples format is now supported.

Refactoring

- Rewriting of the tree-witness rewriter (now independent from Datalog)

Bug fixes

- Protégé: fixed issues related to prefix management.
- Reasoning: issue related to the handling of inverse properties fixed.

3.0.0 (August 23, 2019)

Changed behaviors

- The ``rdfs:range`` of a data property is not used anymore for inferring the non-specified datatype in a mapping assertion but for validating the datatype once specified or inferred. Datatype inference is now based on the DB datatype and follows the R2RML natural mapping (<https://www.w3.org/TR/r2rml/#natural-mapping>)

New features

- Support of Protege 5.5
- Support of BNode in R2RML
- Basic support for the `geo:wktLiteral` datatype (#256)
- New command-line interface for SPARQL endpoint
- Docker container for SPARQL endpoint
(<https://hub.docker.com/r/ontop/ontop-endpoint>)
- Preliminary support of Dremio
- Preliminary support of Denodo
- The ordering of SPARQL variables is now preserved (#291)
- N-Triples output for materialization added
- The SQL translation now appears as a sub-tab in the Protégé Ontop SPARQL tab

Refactoring

- The triple predicate is now systematically used in the mapping and during query unfolding
- Atom predicates and function symbols are now clearly distinguished
- The Datalog-based data structure is not used anymore at mapping processing time and by the default rewriter
- Query containment-based optimizations applied to the mapping after saturation improved
- The materializer is now based on RDF4J (instead of OWLAPI)

Project organisation

- Release of Ontop on both SourceForge and GitHub (#274)
- A new repository for the tutorial
(<https://github.com/ontop/ontop-tutorial>)

Many bugfixes.

3.0.0-beta-2 (February 28, 2018)

Changed Behaviours

- The JDBC driver class name is now required in Protégé (issue #248)

New features

- Novel left join optimization techniques (issue #230)
- LIMIT appended to test SQL queries and SPARQL queries in Protégé
- Report illegal usage of Fully qualified column names (FQDN) in the target part of a mapping assertion
- Support for the SPARQL isNumeric function

Refactoring

- Internal representation of the ontology improved
- Improvement of the intermediate query representation (issues #235, #236)
- Switch to ANTLR4 to parse ontop native mapping (issues #238, #239)
- Make ResultSet.next() robust (issue #219)
- More robust handling of IRI-safe encoding (issue #245)

Bug Fixes

Many, see [closed issues](#) .

3.0.0-beta-1 (August 11, 2017)

Note that this is the first release from version 3 series with many internal refactoring and improvements, and bug fixes. However, these changes also cause incompatibility with Ontop v1. See [the migration guide](#).

Changed Behaviours

- JDBC connection information is stored in a separate properties file
- Fully qualified columns (table.column) are not allowed in the target part of the mapping language. One has to explicitly create an alias (e.g. table.column AS table_column) and use the alias in the target. (complying with R2RML)
- The rules for datatype type inference and checking are changed. Now we use only mapping and database to infer the datatypes (complying with R2RML), and the (range axioms) in the ontologies

to check the compatibility (complying with OWL).

New features

- Major refactoring of the internal structure of the Ontop system
- Databases for Integration Tests have been dockerized.
- Upgrade Sesame to the latest version of RDF4J (v2.2.2).
- The redesigned OWLAPI and RDF4J API bindings of Ontop API (see examples: <https://github.com/ontop/ontop-api-examples>)
- Upgrade of R2RML API to v0.6.0
- Official support of both Tomcat and Jetty (v9) for deploying a SPARQL endpoint
- Better support of Optional (bugfix #190)
- Protege: support for "Synchronize Reasoner" in the Reasoner menu
- Improved error messages for incorrect mapping
- CLI tool for converting Mapping from v1 to v3

1.18.1 (June 16, 2017)

- Return a correct extension for files in ontop cli materializer (issue #145)
- Materialize support non-http resource uris
- Bug Fixes
 - Small fixes in cli (issues #127, #141, #188)
 - PostgreSQL JDBC driver fix for quotation marks
 - Remove unknown property warning for owl:sameAs
 - Remove duplicate annotations during materialization (issues #183 #186)
 - Generate different file names for properties and classes during materialization with separate files (issue #165)
 - R2RML
 - Small changes in api
 - Fix R2RML import of SQL query (issues #181 #182)
 - Fix export from OBDA mapping to R2RML (issues #158 #159 #160)
 - Fix in SPARQL use of filter and values with a IRI (issue #191)

- Fix handling of mysql metadata with lower_case_table_names setting on (issue #192)

1.18.0 (May 26, 2016)

- Protege
 - Redesigned materialization gui
 - Upgrade to Protege 5.0.0
- Support of querying annotation property from ontology (issue #105) (disabled by default)
- Support of the use of owl:sameAs in mappings (disabled by default)
- Package names have been unified to it.unibz.inf.ontop
- Upgrade OWLAPI to v4.2.5
- Bug Fixes
 - Fix obda-to-r2rml convert for template like <{uri}> (issue #97)
 - Fix an issue in Protege when renaming data source.
 - Fix error messages in protege mapping assistant panel and SPARQL query
 - Fix issue with multiple schemas (issue #99)
 - Fix internal SQL conversion for Teiid jdbc driver
 - Fix support of anonymous ontology (issue #101)
 - Fix warning message in Protege (issue #102)
 - Fix xml uri (issue #104)
 - Fix r2rml api, error predicateMap with template (issue #107)
 - Fix in Protege import of additional ontology and add classes and properties of imported ontologies at runtime (issues #106 #108)
 - Fix querying URI constants (issue #111)
 - Fix error message creating new mapping (issue #120)

1.17.1 (February 16, 2016)

- Bug Fixes
 - Fix an issue when creating new datasource in Protege

1.17.0 (February 4, 2016)

- Protege
 - Ontop protege plugin now available on official Protege Plugin Repository!
 - Upgrade Protege to 5.0.0-beta-21 (latest version)
 - Ontop protege plugin is packed as one jar now -- no need for separated OSGI JDBC configuration jars
 - do not generate .obda and .q file when no datasource is present
 - General improvement of UI -- more intuitive and less confusion
- Support for Annotation Properties in mappings
- Improvement of Ontop mapping language for URIs/Literals retrieved from DB columns
- Support for SPARQL VALUES keyword for SPARQL federation
- Java 8
- Upgrade OWLAPI to v4.1.3 (latest version)
- Bash Completion for command line interface
- Better structured Wiki: <https://github.com/ontop/ontop/wiki>

1.16.1 (November 16, 2015)

- Optimizations exploiting foreign key constraints over multiple columns
- Improvement in the RDB2RDF compliance test (contributed by Evren Srin)

1.16.0 (October 14, 2015)

- Support for [SPARQL functions](#)
- Support for column oriented RDBMS [MonetDB](#)
- Support for cloud RDBMS [SAP HANA](#)
- New Optimization technique: [Implicit-database-constraints](#)
- New Optimization technique: [Exact-Mappings](#)
- Bug Fixes
 - Handling quotes in the WHERE clause of SQL query in mappings .
 - Allow multiple subjects in the target of mapping of Ontop native syntax.
 - Return correct value from datetime columns using jtds driver
 - Fix the translation of LIMIT with ORDER BY in oracle database

- Fix a bug in MetaMappingExpander

1.15.0 (May 13, 2015)

- Upgrade the Protege plugin to Protégé 5
- New Command line interface
- Better support for SPARQL BIND
- Support for SPARQL Concat and Replace
- Support for SQL Concat and Replace (in the mappings)
- Support for xsd:dateTimeStamp
- Improved Optimization techniques (Contribution by Roman Konchakov)
- Implementation of Autoclosable interface for core classes in the API to support `try with resource``.
- Create a sub-query when SELECT DISTINCT is used in the mappings.
- Bug Fixes
 - Handle the 30-characters limit for variable names in Oracle.
 - Fix the mapping render of Protege when multiple subjects are in the target of mapping.

1.14.0 (November 4, 2014)

- Added support in ontology and mappings for datatypes xsd:int, xsd:long, xsd:positiveInteger, xsd:negativeInteger, xsd:nonPositiveInteger, xsd:nonNegativeInteger, xsd:float
- Optimized implementations of TBox classifications, and query containment checking, which are much faster than before
- Validation for Sesame Workbench made stronger. It prevents repositories with an invalid configuration to be created.
- Bug Fixes:
 - Fix SPARQL functions (str(), lang()) to use rdfs:Literal instead of xsd:string
 - Issue converting datatypes of R2RML constants
 - Support UNICODE characters in TurtleParser
 - Catch exceptions for Protege R2RML Export
 - Fix conversion from ontop mappings to R2RML of language tag and triples

1.13.1 (September 29, 2014)

- Bug Fixes:
 - The redundant query elimination optimization sometimes does not handle facts properly
 - Some warning messages related to datatype range axioms are ambiguous

1.13.0 (September 25, 2014)

- Support for providing implicit DB Constraints (e.g. foreign keys) to Ontop to generate efficient SQL
- Command line feature to generate pretty R2RML
- Stream output in the Protege Plugin
- Support for Regex in mappings
- Proper handling of datatypes in ontology and mappings
- Support for hsql database
- Support for datatypes xsd:date, xsd:time and xsd:gYear
- Bug Fixes:
 - Issue with nested equivalences in unfold
 - Problem with multiple join conditions in R2RML
 - Cancelling running queries in Protege
 - Blank lines in generated sql causes problems for oracle

1.12.0 (June 26, 2014)

- Support for finding empty concepts and properties
- Integration with a new R2RML API
- Less restrictive mode for JDBC driver validation
- Faster TBox reasoner implementation
- Optional "Replace" Statement in the SQL translation. When "Replace" is off the SQL performs much better.
- Several Bug Fixes:
 - Issue with 'true' in Oracle
 - Meta-mapping editing in Protege
 - Problem with regex in Oracle
 - Several issues with R2RML

1.11.0 (February 19, 2014)

- Support for Consistency Checking
- Improved Performance
- Support for Multi-schema queries
- New SQL parser for the mapping based on JSQLParser.
- Extended the supported SQL language: IN, BETWEEN, LIKE, etc.
- Extended the supported SPARQL: BIND
- Improved the Treewitness rewriting algorithm.
- Upgraded to Sesame 2.7.10
- General clean-up
- Fixed a number of bugs:
 - Translation of SPARQL REGEX into SQL
 - Infinite loop triggered by some nested Optionals
 - Several issues with Construct
 - Several issues with R2RML

1.10.0 (December 16, 2013)

- Ontop under Apache license
- Ontop packages in central maven repository

1.9.0 (September 18, 2013)

- Update to Sesame 2.7.6 (latest)
- Validation of datatypes w.r.t. XMLSchema specification
- New hybrid Abox functionality support
- Removed all dependencies to Jena
- Removed wrong/unnecessary dependencies
- Turtle parser can handle variables in the predicate
- Mappings with URI templates in class or properties support
- First version of mapping splitter
- Bug fix: Null pointer exception when the URI in Protege plugin was empty
- Bug fix: Literal and literal with language tag handling
- Bug fix: Shutdown of Sesame repoConnection
- Bug fix: Object column literal generation in R2RML
- Bug fix: Queries that would not finish execution

- Bug fix: Unexpected bindings in unfold
- Bug fix: Null pointer exception in ontology translation in the case of unsupported axiom
- Bug fix: OWL class assertion generation, inverted arguments
- Bug fix: SameTerm in sparql algebra to datalog translation
- Bug fix: URI template matcher to find most specific matches first
- Bug fix: Construct queries
- Bug fix: Describe queries

1.8.0 (July 29, 2013)

- Improve error message and better exception handling.
- Change the representation of `null` value in BindingSet
- Modified the computation of the LeftJoin to support empty evaluation in its second argument.
- Changed the type of the semantic index table for integer data property assertion from INT to BIGINT to be able to handle `xsd:integer` type
- Set index creation during Quest startup
- Implement new feature for bootstrapping, command line scripts, integration in ontoPro plugin
- Updated the URI templates in Turtle mapping syntax to make it simpler and closer to Turtle. Now quotes, question marks and class/property declarations are no longer required

For example, in the previous syntax:

```
<"test;person/{?id}"> a :MyClass; :hasName {?name}.
```

becomes simply

```
test:person/{id} a :MyClass ; :hasName {name} .
```

- Support "%", "=", ";" characters in the URI template.

```
:class/{id};{name}%{class type}
```

- Bug fix: Major bugs in execution handling SPARQL Construct and

SPARQL Describe queries.

- Bug fix: Issues related to commits and batch sizes in QuestOWLStatement
- Bug fix: Issue with URI template matcher that generates `null` in case failed template (now it generates `URI("xxx")` where xxx is the URI that was being matched.)
- Bug fix: Issue in existential clause in tree-witness algorithm
- Bug fix: Issue in the maximum size of URI that has more than 400 characters
- Bug fix: Issue in some data that can not be tracked when inserted into the database. In particular if the data comes from classes/properties that were simplified by the equivalence resolution.
- Bug fix: Issue in exception handling during index creation in semantic index test.
- Bug fix: Issue in SQL parser that it doesn't capture the scope within parentheses in the WHERE clause.
- Remove table and column identifier quotes in the SQL target query string so that the parser can parse properly. For example:

```
SELECT "Student"."sid" from "Student"
```

will become:

```
SELECT Student.sid from Student
```

- New semantic index implementation with integer references for URIs for faster query answering (faster joins on integers instead of strings)

Ontop source code

- Ontop is now in GitHub!
- Cleaner and slimmer project structure. We removed obsolete classes and tests.
- Code refactor in Materialiser API, MinimalCQProducer and Construct-Describe query execution.
- More complete test cases for testing Quest features, e.g., Tree-Witness rewriting, Data materialiser, SQL parser, SPARQL

Construct, SPARQL Describe, SPARQL Optional, Database metadata and Query completeness.

- Support JUnit4 test cases.

1.7-alpha2 (rev 2305, April 5, 2013)

- Support for importing/exporting R2RML mappings
- Support for automated mapping generation as defined by the "Direct Mapping" W3C specification
- Various options for data materialization (similar to dump-rdf in D2RQ)
- Support for concurrent queries in the SPARQL endpoint through connection pooling
- Better support for Oracle, Postgres, SQL Server and DB2
- Improved the SQL queries generated by Quest by avoiding some operations when unnecessary (e.g., casting), and placing optimally the JOIN conditions in the SQL query.

1.7-alpha2 (rev 2178, February 7, 2013)

-ontopQuest-

- Mapping optimiser: improved the generation of OR mappings
- Mapping optimiser: improved the simplification of mappings using FOREIGN KEYS
- Query optimiser: Improved the query optimiser to avoid generation of redundant JOINS and redundant UNIONS w.r.t. FOREIGN KEYS
- Support percent encoding in strings in URI templates as defined by the R2RML standard.
- Better propagation of Exception messages in the Sesame provider
- New scripts for querying quest from the command line

-ontopPro- plugin

- Introducing mapping assistant. A new GUI-based tool for creating mappings. This is an experimental GUI to help you create mappings in a faster way, please help us improve it by reporting any issues you may find. (See [guideliness](#)).
- Users can set the max number of tuples returned in "OBDA

Query" tab.

Mapping language features (-ontopQuest- and -ontopCore-)

- Support URI identifier obtained directly from the database (no using URI template).
- Support multiple lines when wrting SQL source query (See [feature notes](#))

Bug fixes

- Bug fix: Issue with MySQL and UTF8 characters (Korean in particular)
- Bug fix: Issue with "unbound variable freshX_X found while generating SQL" fixed.
- Bug fix: Inverse minimum cardinality assertion was being ignored.
- Bug fix: Source SQL query doesn't accept parenthesis in WHERE clause.
- Bug fix: JDBC connection doesn't recover when connection lost.
- Bug fix: Sesame virtual repository fails to read Quest preferences.
- Bug fix: CSV export function only exports 101 tuples instead of all the tuples.
- Bug fix: Null pointer exception in the query result table.

1.7-alpha (rev 2039, December 30, 2012)

SPARQL support

- Support for most SPARQL built in functions, e.g., arithmetics, type checking, etc.
- Support for CONSTRUCT and DESCRIBE result forms
- Support for OPTIONAL and bound, which allow you to have NEGATION in the usual SPARQL 1.0 style.
- Support for variables in ANY position, e.g., to allow queries such as `SELECT ?p ?v WHERE { ?x :name "John Smith". ?x ?p ?v }`
- Stronger compliance to standard RDF and SPARQL semantics for FILTER expressions

Performance

- Multiple improvements to the SPARQL-to-rule translation, mapping and TBox optimization algorithms.

- New algorithm for query rewriting w.r.t. the TBox that increases the performance of query answering by many orders of magnitude when inference w.r.t. to existential constants is enabled. The implementation has been contributed by Dr. Roman Kontchakov.
- New benchmarks. We tested against Virtuoso triple store, Virtuoso RDF views and D2R Server using the BSBM and FishMark benchmarks.

APIs and Accessibility

- Completely new Sesame API provider, now users of Sesame can create "Virtual RDF repositories" with -ontop-
- SPARQL end-point support through integration with Sesame's workbench

Mapping language

- Improved support for default literal types, now it conforms with the R2RML "Natural Mapping" specification
- Support for language tags obtained from the DB
- Proper support for constants in the mappings
- Proper internal handling of NULL values in the database (as recommended by the R2RML standard)

Database support

- Restored support for Oracle, DB2 and SQL Server
- Improved support and performance over MySQL

Documentation

- Revised and extended all technical documentation

1.7-alpha-rev-1692 (August 16, 2012)

-ontop- general

- New file content for the OBDA output file. The file is no longer in XML format but instead it uses plain text for simplicity (Issue: #244).
- Improved error messages when the system initializes, parses mappings, fails to

execute queries, etc.

-ontopPro- Protege plugin

- Support Protégé 4.2
- Add cancel button during query execution to terminate the process immediately.
- Add help button to guide filling data source parameters.
- The plugin no longer introduces prefix definitions
- Implemented "Cancel" function for query execution
- New shortcut CONTROL + T to test mappings in the mapping creator
- Fixed several issues with prefix management and rendering of

URIs short forms

- Improved GUI, hints and suggestions for JDBC connections, improved TAB navigation

in all dialogs

- Bug fix: Improve error reporting and stack trace printing (Issue: #255, #262)
- Bug fix: The target and source string got saved partially if newlines are used.
- Bug fix: The OBDA model doesn't get loaded if the query has an exception.
- Improved error management during .obda and query file load
- Fixed null pointer exception and blank window that appear in cases where the .owl file

in OWL/RDF contained different default ns and a default prefix

-ontopQuest- OWL Reasoner

- Added support for SPARQL query modifiers: ORDER BY, OFFSET and LIMIT (Issue: #261).
- Add query caching to speed up the system performance, now repeated query execution will not require SQL generation every time.
- Relaxed on punning checks to allow loading of DBPedia and

similar ontologies.

- EXPERIMENTAL: Added support for mixed ABoxes for class assertions, i.e., most data comes from the mappings but a bit of data is also allowed in the ABox, e.g., global facts `Planet('mars')`
- Improved support for MySQL, Postgres, DB2 and SQL Server dialects
- Added a warnings when axioms are being ignored because of being out of OWL 2 QL
- Improved error reporting on failed connections
- Improved support for SQL AS for table and column renaming in mappings
- New way to generate SQL. SQL now includes typing information and is pretty printed.
- Bug fix: Fixed issue with INVALID URI that appeared when mappings used data that contained invalid characters for URI's (e.g., white space) Now we call `URIEncode` (Jena's) before returning the value.
- Bug fix: Exception when importing data as ABox individuals.
- Bug fix: Exception when null values is used as constants.
- Bug fix: Exception if ambiguous column names are used in SQL queries.
- Bug fix: Wrong URI shorten if there are two or more namespaces with overlapping strings.
- Bug fix: Null pointer exception due to inconsistent OBDA model identifier string.
- Bug fix: Unknown signature in the SELECT clause if the name is a qualified name and it comes from a view (Issue: #259)
- Bug fix: Exception when using URIs as a constant values in SPARQL.
- Bug fix: Prefix Manager accepts any string including invalid prefix names.

Known issues:

- The ORDER BY modifier is not ready for columns with literal datatype
- Issues with MySQL in non-ANSI mode
- DateTime data properties in Oracle can crash the system.

1.7-alpha-rev-1529 (June 12, 2012)

-ontopPro- plugin

- Better layout, fonts and command icons.
- Better query saving mechanism (Issue: #248)
- Better error reporting to users and crash handling (Issue: #255, 256)
- Pretty printing of SQL queries (Issue: #254)
- Re-enabled the export function in the Query Interface panel (Issue: #213, #250)
- Improved syntax-error messages in the mapping editor tooltip
- Cleaned console messages
- Added "Number of sources" indicator to show how many sources are already declared in the OBDA model.
- Improved ESC and CONTROL+ENTER shortcuts in most dialogs in the GUI.
- Replaced JDBC Driver text box with an editable Combo Box with default driver string.
- Better OBDA statistics table (Issue: #251)
- Bug fix: CRITICAL stability issue that would rise an end-less cycle of warning messages when Quest crashed.
- Bug fix: issue with final dot "." in mappings.

-ontopQuest-

- Added support for data-type UUIDs for SQL Server (Issue: #253)
- Now we create forward-only and concur-read-only JDBC Statements by default to allow for server side cursors.
- Bug fix: issue with SQL parser and table renamings in SQL queries (Issue: #259)
- Bug fix: issue with NULL pointer when primary key optimization in unfolding when there is a join on primary condition but atoms are not unifiable.

Known issues:

- Oracle is unstable

1.7-alpha-rev-1479 (May 22, 2012)

Quest performance and features:

- T-Mappings algorithm optimization: Combine disjunctive conditions in OR clauses instead of multiple mappings. This reduces the size of the final query and improves its performance. (Issue: #237)

Mapping language features (Quest and ontopCore):

- Support using IS NULL and IS NOT NULL in SQL query (Issue: #200).
- Support using UNION in SPARQL queries.

Bug Fixes:

- Fix packing problem in Windows XP (Issue: #228)
- Fix packing problem in Linux
- Fix unresponsive plugin when using mapping filter (Issue: #217).
- Fix unresponsive plugin when inserting a new mapping (Issue: #232).
- Fix support for MySQL in both virtual and classic mode (Issue: #238).
- Fix support for Oracle in virtual mode (Issue: #226).
- Fix issue where creating or modifying a mapping would not trigger a "ontology out of synch" flag.

Known Issue:

- Oracle still has some issues in classic mode.

1.7-alpha-rev-1446 (May 4, 2012)

Mapping language features (Quest and ontopCore):

- Add support on writing qualified name for atom predicates.
- Add coloring for atoms typed Individual in the target query.

Bug Fixes:

- Fix the "Show Short URIs" feature in the query panel.

1.7-alpha-rev-1440 (May 2, 2012)

Mapping language features (Quest and ontopCore):

- Support writing URI constants in target queries (Issue: #227)

General:

- Better system logging for recording exceptions.

Bug Fixes:

- Fix the null pointer exception due to the old OBDA preferences.
- Fix bugs related to vocabulary prefixes.

1.7-alpha-rev-1427 (April 17, 2012)

Quest performance and features:

- Implemented the T-Mappings mapping optimization algorithm improved performance of query rewriting in Virtual Mode. Now query rewriting is as fast in virtual ABox mode as in classic ABox mode (Issue: #169)
- Implemented a new query containment algorithm that is many times faster than before.
- Added data source analysis to obtain DB metadata for SQL query generation.
- Added analysis of SQL queries in the mappings to produce better SQL queries.
- Added SQL JOIN ELIMINATION using PRIMARY KEY metadata.
- Added mappings optimization using query containment w.r.t. constraints
- Added initial support for FILTER
- Added support for typed constants in SPARQL queries (Issue: #157)
- Small updates to internal API for speed of equality, hash and clone computation

Mapping language features (Quest and ontopCore):

- New Turtle-like syntax for mappings allows for more succinct and easy to read mappings.
- Added support for LANGUAGE tag for the Literal datatype (Issue: #198)

- Added support for datatypes in the OBDA mapping axioms (Issue: #147, #194, #196)
- Added URI templates for URI creation in mapping axioms (Issue: #151)
- New feature for typing literals in mapping axioms (Issue: #194, #207)

General:

- Added support for OWL API 3 (Issue: #98)
- Internal refactoring of all modules and packages for a cleaner API

ontopPro:

- The plugin is now compatible with Protege 4.1 (Issue: #99)
- The plugin has a new and fresh GUI display.
- Stronger integration with Protege's refactoring mechanism
- The ABox materializer can import data complete with datatype information (Issue: #195)
- The target query use Turtle syntax for better display and management (Issue: #204)
- The plugin has Class/Property panel for better navigation and filtering.

Bug Fixes:

- Fixed a problem with the encoding of axioms of the form `subClassOf(Person, hasFather some Person)`
- Fixed issues with open and dead JDBC connections
- Cleaned OWLReasoner synchronization issue
- Fixed null pointer exceptions related to incorrect mappings
- Proper error handling for translating an empty datalog program (Issue: #197)
- Fixed the "Attach Prefixes" feature in the OBDA Query panel (Issue: #211)
- Fixed issue regarding `xsd:dateTime` (Issue: #199)

1.6.1 (October 19, 2011)

- Fast patch in the query translation module.

1.6 (October 18, 2011)

- Allow fast navigation using TAB key in the new mapping form (Issue: #92)
- Allow SQL query execution in the mapping tree (Issue: #104)
- Improvement in the error message deliveries (Issues: #107, #112, #118)
- Improvement in the data source management tab, including adding "Test Connection" button (Issues: #108, #111, #114)
- Add OBDA model statistics report (Issues: #106, #117)
- Add support for adding integrated data source in Teiid (Issue: #89)
- Fix the unresponsive cancel button during importing OWL Individuals (Issue: #88)
- Fix CSV header in the exported query result (Issue: #120)
- Fix connection failure when users change the parameters (Issue: #121)
- Fix the password field so now the text is hidden (Issue: #113)
- Fix query selection in the OBDA query tab (Issue: #91)
- Fix import failure when storing very long record values (Issue: #95)
- Fix import failure when storing records that contain single quotes (Issue: #96)

1.5.1-RC1 (July 15, 2011)

- Significant code refactoring in the plugin. This effort makes the code easier to debug and to extend, and is almost ready for Protege 4.1 and OWL-API 3.0 support.
- Term renaming. Modified the term 'Mapping Head' to 'Target Query' and 'Mapping Body' to 'Source Query' to maintain the consistency with the literature.
- Temporarily disabled the SPARQL query coloring.
- Temporarily disabled the 'Load OWL Individuals into DB' functionality.
- Add improved search filters in the mapping manager (Issues: #37, #54).
- Add 'show short URIs' option in the query results to hide long URI strings.
- Add 'select all', 'select none', 'expand all' and 'collapse all' buttons

for easy browsing in the mapping tree (Issues: #35, #36).

- Add mapping validation with respect to the vocabulary in the ontology (Issues: #42).
- Add input validation to many parts of the GUI (Issues: #63, #64, #65, #67, #68, #69, #70, #71, #76).
- Fix the save query functionality in the Query manager (Issues: #57).
- Fix the persistency of user preferences when opening/closing Protege (Issues: #58).
- Fix the exceptions that appeared when loading/saving/reloading ontologies (Issues: #59, #60, #61, #75).
- Fix problems with several progress dialogs in the GUI (Issues: #50, #87).
- Fix problems in the 'tuple count' functionality (Issues: #82, #84).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Glossary](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance
 - SPARQL 1.1
 - GeoSPARQL 1.0
 - R2RML
 - RDF 1.1
 - OWL 2 QL
 - RDFS
 - Time functions
 - Other functions

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Standards compliance

SPARQL 1.1

Updated for 5.4.0.

In the following table we present a summary of the compliance of the latest version of Ontop with SPARQL 1.1 , where rows correspond to sections of the W3C recommendation and unsupported features are ~~crossed out~~. Most of the features are supported, but some are unsupported or only partially supported.

Section in SPARQL 1.1	Features	Coverag
5. Graph Patterns	BGP , FILTER	2/2
6. Including Optional Values	OPTIONAL	1/1
7. Matching Alternatives	UNION	1/1
8. Negation	MINUS , FILTER [NOT] EXISTS	2/2
9. Property Paths	PredicatePath , InversePath , SequencePath , AlternativePath , NegatedPropertySet , ZeroOrMorePath , OneOrMorePath , ZeroOrOnePath	5/8
10. Assignment	BIND , VALUES	2/2
11. Aggregates	COUNT , SUM , MIN , MAX , AVG , GROUP_CONCAT , SAMPLE	6/6
12. Subqueries	Subqueries	1/1

13. RDF Dataset	GRAPH [NAMED] FROM	2/2
14. Basic Federated Query	SERVICE	0
15. Solution Seqs. & Mods.	ORDER BY DISTINCT OFFSET SELECT REDUCED LIMIT	6/6
16. Query Forms	SELECT ASK CONSTRUCT DESCRIBE	4/4
17.4.1. Functional Forms	BOUND COALESCE NOT EXISTS && IN IF EXISTS = sameTerm NOT IN	11/11
17.4.2. Functions on RDF Terms	isIRI isBlank isLiteral isNumeric lang IRI STRDT UUID str datatype BNODE STRLANG STRUUID	11/13
17.4.3. Functions on Strings	STRLEN UCASE STRSTARTS STREND STRBEFORE STRAFTER ENCODE_FOR_URI CONCAT langMatches REGEX SUBSTR LCASE CONTAINS REPLACE	14/14
17.4.4. Functions on Numerics	abs ceil round floor RAND	5/5
17.4.5. Functions on Dates&Times	now month hours seconds tz year day minutes timezone	8/9
17.4.6. Hash Functions	MD5 SHA256 SHA512 SHA1 SHA384	5/5
	xsd:string	

17.5 XPath Constructor Functions	xsd:float xsd:double xsd:decimal xsd:integer xsd:boolean xsd:dateTime	7/7
17.6 Extensible Value Testing	~~user-defined functions~~	0

Limitations

- The 5 hash functions and functions `REPLACE` and `REGEX` for regular expressions have limited support because they heavily depend on the DBMS: not all DBMSs provide all hash functions, and many DBMSs have their own regex dialects. Currently, the SPARQL regular expressions of `REPLACE` and `REGEX` are simply sent to the DBMS by default.
- In the implementation of function `langMatches`, the second argument has to be a constant: allowing variables will have a negative impact on the performance in our framework.
- Cast to `xsd:boolean`. Casting NaN to boolean should result in `"false"^^xsd:boolean`, however this check is not feasible for many DB dialects, hence there is mostly no special handling of these values with support limited to PostgreSQL, Oracle and Spark.
- Cast to `xsd:decimal`. Casting to DECIMAL results in values with scale 0 in several SQL dialects. As such the custom DECIMAL_STR with an arbitrary precision defined in the respective DBTypeFactory of a dialect is used for each cast.
- Cast to `xsd:dateTime`. The list of possible timestamp patterns is too large, and many dialects lack an effective way to check for these patterns without getting extremely verbose. Hence, it is not possible to often detect whether a value is a valid timestamp before the cast is performed. With the exception of SQL Server and Snowflake which have TRY_CAST functions, applying this function to other dialects comes with severe limitations.
- `EXISTS` is only supported when it can be translated into a

bottom-up manner, i.e. using a left join. In practice, it covers most of the cases, but sometimes a top-down evaluation is necessary.

GeoSPARQL 1.0

Starting from 4.1.0.

The following table provides a summary of the compliance of the latest version of Ontop with [OGC GeoSPARQL 1.0](#) , the standard for representing and querying geospatial linked data. The summary focuses only on the main geospatial functions and properties and unsupported features are ~~crossed out~~.

Section in OGC GeoSPARQL 1.0	Features	Co
7. Topology Vocabulary Extensions - Properties	geo:sfEquals geo:sfDisjoint geo:sfIntersects geo:sfTouches geo:sfCrosses geo:sfWithin geo:sfContains geo:sfOverlaps geo:ehEquals geo:ehDisjoint geo:ehMeet geo:ehOverlap geo:ehCovers geo:ehCoveredBy geo:ehInside geo:ehContains geo:rcc8eq geo:rcc8de geo:rcc8ec geo:rcc8po geo:rcc8tppi geo:rcc8tpp geo:rcc8ntpp geo:rcc8ntppi	0
8.4. Standard Properties for Geo:Geometry	geo:dimension geo:coordinateDimension geo:spatialDimension geo:isEmpty geo:isSimple	0

	<code>geo:hasSerialization</code>	
8.5. WKT Serialization	<code>geo:wktLiteral</code> <code>geo:asWKT</code>	2/2
8.6. GML Serialization	<code>geo:gmlLiteral</code> <code>geo:asGML</code>	0
8.7. Non- Topological Query Functions	<code>geof:distance</code> <code>geof:buffer</code> <code>geof:convexHull</code> <code>geof:intersection</code> <code>geof:union</code> <code>geof:difference</code> <code>geof:symDifference</code> <code>geof:envelope</code> <code>geof:boundary</code> <code>geof:getSRID</code>	10/
9.2. Common Query Functions	<code>geof:relate</code>	1/1
9.3. Topological Simple Features Relation Family Query Functions	<code>geof:sfEquals</code> <code>geof:sfDisjoint</code> <code>geof:sfIntersects</code> <code>geof:sfTouches</code> <code>geof:sfCrosses</code> <code>geof:sfWithin</code> <code>geof:sfContains</code> <code>geof:sfOverlaps</code>	8/8
9.4. Topological Egenhofer Relation Family Query Functions	<code>geof:ehEquals</code> <code>geof:ehDisjoint</code> <code>geof:ehMeet</code> <code>geof:ehOverlap</code> <code>geof:ehCovers</code> <code>geof:ehCoveredBy</code> <code>geof:ehInside</code> <code>geof:ehContains</code>	8/8
9.5. Topological RCC8 Relation Family Query Functions	<code>geof:rcc8eq</code> <code>geof:rcc8dc</code> <code>geof:rcc8ec</code> <code>geof:rcc8po</code> <code>geof:rcc8tppi</code> <code>geof:rcc8tpp</code> <code>geof:rcc8ntpp</code> <code>geof:rcc8ntppi</code>	8/8

Several non-topological query functions use a unit of measure

URI which OGC defines under a specific namespace e.g.

`<http://www.opengis.net/def/uom/OGC/1.0/metre>`. The

latest version of Ontop currently supports the units metre, radian and degree.

R2RML

Updated for 4.2.0

The latest version of Ontop is almost fully compliant with the [R2RML](#) standard.

At the moment, it does NOT support:

- Base IRIs
- [R2RML default mapping](#) generation
- Normalization of binary SQL datatypes

For complex SQL queries (e.g. with a `GROUP BY`) in the mapping, Ontop may not be able to infer the datatype of each column if the option

`ontop.allowRetrievingBlackBoxViewMetadataFromDB` is not

enabled (disabled by default). In such a situation, it may not be able to derive the [natural RDF datatype](#) of a literal built over a column and may not apply the expected normalization. This can be partially mitigated by expliciting the RDF datatype in the mapping, but normalization would remain unapplied.

RDF 1.1

Ontop complies with [RDF 1.1](#). It types simple literals (from RDF 1.0) as `xsd:string` and literals with a language tag as

`rdf:langString`.

OWL 2 QL

RDFS

Time functions

Updated for 5.1.0.

The functions using the prefix `ofn` (`http://www.ontotext.com/sparql/functions/`) and their documentation can be found [here](#) . They accept both `xsd:date` and `xsd:dateTime` as arguments. (supported since 4.2.0).

The functions using the prefix `obdaf` (`https://w3id.org/obda/functions#`) have been introduced in 5.1.0 (see [#705](#)).

Function	Argument 1	Argument 2
<code>ofn:weeksBetween</code>	<code>xsd:date</code>	<code>xsd:date</code>
<code>ofn:weeksBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:weeksBetween</code> *	<code>xsd:date</code>	<code>xsd:dateTime</code>
<code>ofn:weeksBetween</code> *	<code>xsd:dateTime</code>	<code>xsd:date</code>
<code>ofn:daysBetween</code>	<code>xsd:date</code>	<code>xsd:date</code>
<code>ofn:daysBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:daysBetween</code> *	<code>xsd:date</code>	<code>xsd:dateTime</code>
<code>ofn:daysBetween</code> *	<code>xsd:dateTime</code>	<code>xsd:date</code>
<code>ofn:hoursBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:minutesBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:secondsBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:millisBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>obdaf:dateTrunc</code>	<code>xsd:dateTime</code>	<code>xsd:string</code>
<code>obdaf:milliseconds-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:microseconds-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:week-from-dateTime</code>	<code>xsd:dateTime</code>	

<code>obdaf:quarter-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:decade-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:century-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:millenium-from-dateTime</code>	<code>xsd:dateTime</code>	

Combinations of argument datatypes marked with the symbol * are not supported for queries over the following data sources: Oracle and Microsoft SQL Server.

The `obdaf:[datePart]-from-dateTime` functions are supported for all dialects. They can be used to extract a specific part of the provided `dateTime` value in a numeric format (`xsd:decimal` for `milliseconds` and `microseconds`, `xsd:integer` for the remaining functions).

NOTE

The function `obdaf:week-from-dateTime` returns the ISO week index of the given date, where week 1 is considered the first week with a majority of its days in January.

The `obdaf:dateTrunc` function can be used to truncate a given `xsd:dateTime` to a new value with specified granularity. The granularity must be provided as an `xsd:string` literal. The following granularity values are supported:

- `microsecond`
- `millisecond`
- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`

- month
- quarter
- year
- decade
- century
- millennium

WARNING

Not all database systems support all granularities equally.

decade, century, and millennium are not supported by:

- AWS Athena
- Denodo (century is supported)
- MySQL (century is supported)
- MariaDB (century is supported)
- Oracle
- Presto
- SQLServer
- Snowflake
- Spark
- Trino

second is not supported by:

- Denodo

millisecond and microsecond are not supported by:

- AWS Athena
- Denodo
- MySQL
- MariaDB
- Oracle
- Presto
- Trino

PostgreSQL requires these granularities to be named milliseconds and microseconds instead.

Examples

```
obdaf:year("2023-08-16T09:00:00"^^xsd:dateTime)
"2023"^^xsd:integer
```

```
obdaf:hour("2023-08-16T09:00:00"^^xsd:dateTime)
"9"^^xsd:integer
```

```
obdaf:dateTrunc("2023-08-16T09:00:00"^^xsd:dateTime,
"month"^^xsd:string) "2023-08-
01T00:00:00"^^xsd:dateTime
```

Other functions

Updated for 5.5.0.

- Cast to `xsd:date`
- Query ID: `obdaf:queryId`

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Command Line Interface](#) [Ontop Mapping Language →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
 - Setup Ontop CLI
 - ontop endpoint
 - ontop materialize
 - ontop mapping
 - ontop bootstrap
 - ontop query
 - ontop extract-db-metadata
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Command Line Interface

Ontop ships a shell script (`ontop` for *nix) and a bat file (`ontop.bat` for Windows) exposing the core functionality and several utilities through the command line interface. It is an easy way to get the system quickly set-up, test for correct execution, and query or materialize as needed.

- Setup
- ontop endpoint
- ontop materialize
- ontop mapping
- ontop bootstrap
- ontop query
- ontop extract-db-metadata

Setup Ontop CLI

First, you have to download Ontop latest CLI zip from our download pages ([Github](#) or [Sourceforge](#)). Unzip it in a folder. Open the command line terminal and cd to that folder. For Windows use the `ontop.bat` file, for Linux and OS X use the `ontop` file.

```
$ ./ontop help
usage: ontop <command> [ <args> ]

Commands are:
  --version          Show version of ontop
  bootstrap          Bootstrap ontology and
mapping from the database
  endpoint           Start a SPARQL endpoint
powered by Ontop
  extract-db-metadata Extract the DB metadata
and serialize it into an output JSON file
  help              Display help information
```

<code>materialize</code>	Materialize the RDF graph exposed by the mapping and the OWL ontology
<code>query</code>	Query the RDF graph exposed by the mapping and the OWL ontology
<code>validate</code>	Validate Ontology and Mappings
<code>mapping</code>	Manipulate mapping files

JDBC configuration

JDBC drivers are software implemented by third parties (often the same developers of the database system) that handle interaction with the database in their own proprietary protocols. You will need to manually download the JDBC drivers for your database management system (e.g., [PostgreSQL JDBC drivers](#)) and put them into the `jdbc` directory.

PATH

Consider putting the directory of ontop to your `PATH`.

Logging level

It can be set using the environment variable `ONTOP_LOG_LEVEL`. Set it to `DEBUG` to get most details, including generated SQL queries. Alternatively, you can change the value of the `level` attribute of the `root` element at the end of the file `log/logback.xml`.

Property file

Most commands below require or accept as input a property file. This is where you will specify the JDBC connection parameters. A basic property file template can be found [here](#).

ontop endpoint

`ontop endpoint` deploys a SPARQL endpoint locally at the

address `/sparql` and by default on the port 8080. It powers [our official Docker](#), so feel free to use the Docker image instead of the CLI command if it is more convenient for you.

It offers several advanced options:

- *Lazy initialization*: Ontop offline tasks (such as DB metadata extraction and mapping processing) are triggered after receiving the first SPARQL query. This is useful when using a Docker-Compose with Ontop and a DB image that needs to be initialized first.
- *Development mode (since 4.0-beta-1)*: restarts the endpoint every time the configuration files are changed. It also exposes a GET/POST method `/ontop/reformulate` accepting a SPARQL query as param as any SPARQL endpoint but returning the reformulated SQL query as result.
- *Portal (since 4.0-beta-1)*: Includes groups of pre-defined SPARQL queries into the welcome page. See the following [example of portal file](#) in the TOML format.
- *Predefined query endpoint (since 4.1.0)*
- *Ontology made downloadable (since 4.2.0)*.

```
$ ./ontop help endpoint
NAME
    ontop endpoint - Start a SPARQL endpoint
    powered by Ontop

SYNOPSIS
    ontop endpoint [ {-a | --facts} <fact
    file> ]
                    [ {-c | --constraint} <constraint
    file> ]
                    [ --contexts <JSON-LD context file
    for predefined queries> ]
                    [ --cors-allowed-origins <origins>
    ]
                    [ {-d | --db-metadata} <db-
    metadata file> ]
                    [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                    [ --db-url <DB URL> ] [ --dev ] [
    --disable-portal-page ]
                    [ --enable-annotations ] [ --
```

```
enable-download-ontology ]
    [ --facts-base-iri <Base IRI of
facts in fact file> ]
    [ --facts-format <format of facts
file> ]
    [ {-l | --lenses | -v | --ontop-
views} <lenses file> ]
    [ --lazy ] {-m | --mapping}
<mapping file>
    [ {-p | --properties} <properties
file> ] [ --port <port> ]
    [ --portal <endpoint portal file>
]
    [ --predefined-config <predefined
query JSON config file> ]
    [ --predefined-queries <predefined
query TOML file> ]
    [ --sparql-rules <SPARQL rules
file> ]
    [ {-t | --ontology} <ontology
file> ]
    [ {-u | --db-user} <DB user> ]
    [ {-x | --xml-catalog} <xml
catalog file> ]
```

OPTIONS

```
-a <fact file>, --facts <fact file>
    User-supplied constant fact file

-c <constraint file>, --constraint
<constraint file>
    User-supplied DB constraint file

--contexts <JSON-LD context file for
predefined queries>
    File containing JSON-LD contexts for
predefined queries

--cors-allowed-origins <origins>
    CORS allowed origins

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)
```



```

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--dev
    development mode

--disable-portal-page
    Disable the portal page (/index.html)
    of the SPARQL endpoint.

--enable-annotations
    enable annotation properties defined
    in the ontology. Default:
    false

--enable-download-ontology
    Allow to download the ontology as a
    plain text file (/ontology).
    Default: false

--facts-base-iri <Base IRI of facts in
fact file>
    The base IRI used for the facts taken
    from the fact file.

--facts-format <format of facts file>
    The format of the materialized
    ontology. Default: infer from file extension

    This options value is restricted to
    the following set of values:
        rdfxml
        turtle
        ntriples
        nquads
        trig
        jsonld

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
--ontop-views <lenses file>
    User-supplied lenses file. Lenses were
    formerly named Ontop views.

```

```

--lazy
    lazy initialization

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in
    Ontop native format (.obda)

-p <properties file>, --properties
<properties file>
    Properties file

--port <port>
    port of the SPARQL endpoint

--portal <endpoint portal file>
    endpoint portal file (including title
    and queries)

--predefined-config <predefined query JSON
config file>
    predefined query config file

    This option is required if any of the
    following options are
    specified: --predefined-queries

--predefined-queries <predefined query
TOML file>
    predefined SPARQL queries file

    This option is required if any of the
    following options are
    specified: --predefined-config

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)

```

```
-x <xml catalog file>, --xml-catalog <xml
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports
```

Example

```
$ ./ontop endpoint -m university-complete.obda \
    -t university-complete.ttl \
    -p university-
complete.properties \
    --cors-allowed-origins=*
```

ontop materialize

This command provides a "materialization utility". Materialization is helpful when you want to generate RDF data out of your database, using the provided mappings. This utility will take all the triples that the mapping can produce from the data source, and write them to the output. `ontop materialize` does not need any query file, but instead, needs the user to specify a format in which he/she wants the output (either to terminal or output file). The user can choose between three output formats: [Turtle](#) , [N-triples](#) or [RDF/XML](#) . For very large datasets, producing the output might take some time.

The compression option has been added in 5.2.0.

```
$ ./ontop help materialize
NAME
    ontop materialize - Materialize the RDF
graph exposed by the mapping
and the OWL ontology

SYNOPSIS
    ontop materialize [ {-a | --facts} <fact
file> ]
                        [ {-c | --constraint} <constraint
```

```
file> ]
        [ --compression <output
compression> ]
        [ {-d | --db-metadata} <db-
metadata file> ]
        [ --db-driver <DB driver> ] [ --
db-password <DB password> ]
        [ --db-url <DB URL> ] [ --enable-
annotations ]
        [ {-f | --format} <output format>
]
        [ --facts-base-iri <base IRI of
facts in fact file> ]
        [ --facts-format <format of facts
file> ]
        [ {-l | --lenses | -v | --ontop-
views} <Lenses file> ]
        {-m | --mapping} <mapping file> [
--no-streaming ]
        [ {-o | --output} <output> ]
        [ {-p | --properties} <properties
file> ] [ --separate-files ]
        [ --sparql-rules <SPARQL rules
file> ]
        [ {-t | --ontology} <ontology
file> ]
        [ {-u | --db-user} <DB user> ]
        [ {-x | --xml-catalog} <xml
catalog file> ]
```

OPTIONS

-a <fact file>, --facts <fact file>
RDF fact file

-c <constraint file>, --constraint
<constraint file>
User-supplied DB constraint file

--compression <output compression>
The compression format of the
materialized RDF graph. Default: no
compression

This options value is restricted to
the following set of values:

gzip
zip

```

no_compression

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--enable-annotations
    enable annotation properties defined
in the ontology. Default:
    false

-f <output format>, --format <output
format>
    The format of the materialized RDF
graph. Default: rdxml

    This options value is restricted to
the following set of values:
        rdxml
        turtle
        ntriples
        nquads
        trig
        jsonld

--facts-base-iri <base IRI of facts in
fact file>
    The base IRI of facts in the fact file
to resolve relative IRIs. If
    not provided, a random IRI is
generated.

--facts-format <format of facts file>
    The format of the 'facts' input file.

    This options value is restricted to
the following set of values:
        rdxml

```

```

turtle
ntriples
nquads
trig
jsonld

-l <Lenses file>, --lenses <Lenses file>,
-v <Lenses file>,
--ontop-views <Lenses file>
    User-supplied lenses file. Lenses were
    formerly named Ontop views.

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in
    Ontop native format (.obda)

--no-streaming
    All the SQL results of one big query
    will be stored in memory. Not
    recommended. Default: false.

-o <output>, --output <output>
    output file (default) or prefix (only
    for --separate-files)

-p <properties file>, --properties
<properties file>
    Properties file

--separate-files
    generating separate files for
    different classes/properties. This is
    useful for materializing large OBDA
    setting. Default: false.

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)

-x <xml catalog file>, --xml-catalog <xml

```

```
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports
```

Examples

```
$ ./ontop materialize -m university-complete.obda
\
    -t university-complete.ttl \
    -p university-
complete.properties \
    -f turtle \
    -o materialized-triples.ttl
```

In case you have some lenses:

```
$ ./ontop materialize -m mapping.ttl \
    -t ontology.ttl \
    -l lenses.json \
    -p configuration.properties
\
    -f turtle \
    -o materialized-triples.ttl
```

ontop mapping

This command collects several useful sub-commands for dealing with mappings files.

```
$ ./ontop help mapping
NAME
    ontop mapping - Manipulate mapping files

SYNOPSIS
    ontop mapping { pretty-r2rml | to-obda |
to-r2rml | v1-to-v3 } [--]
    [cmd-options]
```

```

Where command-specific options [cmd-
options] are:
    pretty-r2rml: {-i | --input}
<input.ttl> {-o | --output}
               <pretty.ttl>
    to-obda: {-i | --input} <mapping.ttl>
[ {-o | --output} <mapping.obda> ]
    to-r2rml: [ {-l | --lenses | -v | --
ontop-views} <lenses file> ] [ {-t | --ontology}
<ontology.owl> ]
               {-i | --input} <mapping.obda>
[ {-o | --output} <mapping.ttl> ]
               [ {-p | --properties}
<properties file> ] [ {-d | --db-metadata} <db-
metadata file> ]
               [ --force ]
    v1-to-v3: [ --simplify-projection ] {-
m | --mapping} <mapping file>
               [ --overwrite ] [ {-o | --
output} <mapping.obda> ]

See 'ontop help mapping <command>' for
more information on a specific command.

```

ontop mapping to-r2rml

Supports automatically converting mappings from Ontop native format (`.obda`) to the `R2RML` standard format. Since 4.1.0, by default, it expects DB credentials (for extracting the DB metadata) or a DB metadata file. This requirement can be bypassed using the option `--force`.

```

$ ./ontop help mapping to-r2rml
NAME
    ontop mapping to-r2rml - Convert ontop
native mapping format (.obda) to
R2RML format

SYNOPSIS
    ontop mapping to-r2rml [ {-d | --db-
metadata} <db-metadata file> ]
                          [ --force ] {-i | --input}
<mapping.obda>
                          [ {-l | --lenses | -v | --ontop-

```



```
views} <lenses file> ]
    [ {-o | --output} <mapping.ttl> ]
    [ {-p | --properties} <properties
file> ]
    [ {-t | --ontology} <ontology.owl>
]
```

OPTIONS

```
-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--force
    Force the conversion in the absence of
DB metadata

-i <mapping.obda>, --input <mapping.obda>
    Input mapping file in Ontop native
format (.obda)

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
--ontop-views <lenses file>
    User-supplied lenses file. Lenses were
formerly named Ontop views.

-o <mapping.ttl>, --output <mapping.ttl>
    Output mapping file in R2RML format
(.ttl)

-p <properties file>, --properties
<properties file>
    Properties file

-t <ontology.owl>, --ontology
<ontology.owl>
    OWL ontology file
```

ontop mapping to-obda

Supports automatically converting mappings from [R2RML](#) standard format to Ontop native format ([.obda](#)):

```
$ ./ontop help mapping to-obda
```

```
NAME
    ontop mapping to-obda - Convert R2RML
    format to ontop native mapping
    format (.obda)

SYNOPSIS
    ontop mapping to-obda {-i | --input}
    <mapping.ttl>
                        [ {-o | --output} <mapping.obda> ]

OPTIONS
    -i <mapping.ttl>, --input <mapping.ttl>
        Input mapping file in R2RML format
        (.ttl)

    -o <mapping.obda>, --output <mapping.obda>
        Output mapping file in Ontop native
        format (.obda)
```

`ontop mapping pretty-r2rml`

Provides automatic formatting and prettifying facilities for mappings files:

```
$ ./ontop help mapping pretty-r2rml
NAME
    ontop mapping pretty-r2rml - prettify
    R2RML file using Jena

SYNOPSIS
    ontop mapping pretty-r2rml {-i | --input}
    <input.ttl>
                        {-o | --output} <pretty.ttl>

OPTIONS
    -i <input.ttl>, --input <input.ttl>
        Input mapping file in the turtle R2RML
        format (.ttl)

    -o <pretty.ttl>, --output <pretty.ttl>
        Output mapping file in the turtle
        R2RML format (.ttl)
```

ontop bootstrap

This command allows the automatic generation of mappings and ontology starting from a database schema. The generated output can be used as-is or further customized manually (e.g., to used different ontological modeling choices and corresponding mappings). In both cases, it helps substantially reducing the user effort involved in setting up the ontology and mappings of a VKG specification.

```
$ ./ontop help bootstrap
NAME
    ontop bootstrap - Bootstrap ontology and
    mapping from the database

SYNOPSIS
    ontop bootstrap {-b | --base-iri} <base
    IRI>
                    [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                    [ --db-url <DB URL> ] {-m | --
    mapping} <mapping file>
                    [ {-p | --properties} <properties
    file> ]
                    {-t | --ontology} <ontology file>
                    [ {-u | --db-user} <DB user> ]

OPTIONS
    -b <base IRI>, --base-iri <base IRI>
        Base IRI of the generated mapping

    --db-driver <DB driver>
        DB driver (overrides the properties)

    --db-password <DB password>
        DB password (overrides the properties)

    --db-url <DB URL>
        DB URL (overrides the properties)

    -m <mapping file>, --mapping <mapping
    file>
        Output mapping file in the Ontop
        native format (.obda)
```

```
-p <properties file>, --properties
<properties file>
    Properties file

-t <ontology file>, --ontology <ontology
file>
    Output OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)
```

ontop query

The `ontop query` command is designed for helping users to test their system quickly using the command line utilities. You can use this command if you already have a scenario test case including:

- the ontology (RDFS or OWL) and the mappings (obda or R2RML) files,
- a working database to connect to,
- a SPARQL query file

The `ontop query` command helps you to set up the system, runs the query from the query string file over it, and gets the results either in output file or terminal output. What the script actually does is to set up Ontop using the ontology and the mapping files, parse the query from the file and execute it over the created instance of Ontop.

Note that `ontop query` is NOT intended to be used in production and for benchmarking purposes. Most of its execution time is dedicated to offline tasks like DB metadata extraction and mapping processing. Query answering (i.e. answering the SPARQL query) takes usually much less time. For production and benchmarking purposes, please consider [deploying Ontop as a SPARQL endpoint](#).

The results are turned in the CSV format.

WARNING

At the moment only SELECT queries are supported by this command. See #222 .

```
$ ./ontop help query
NAME
    ontop query - Query the RDF graph exposed
    by the mapping and the OWL
    ontology

SYNOPSIS
    ontop query [ {-a | --facts} <fact file> ]
                [ {-c | --constraint} <constraint
    file> ]
                [ {-d | --db-metadata} <db-
    metadata file> ]
                [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                [ --db-url <DB URL> ] [ --enable-
    annotations ]
                [ --facts-base-iri <Base IRI of
    facts in fact file> ]
                [ --facts-format <format of facts
    file> ]
                [ {-l | --lenses | -v | --ontop-
    views} <lenses file> ]
                {-m | --mapping} <mapping file> [
    {-o | --output} <output> ]
                [ {-p | --properties} <properties
    file> ]
                {-q | --query} <queryFile>
                [ --sparql-rules <SPARQL rules
    file> ]
                [ {-t | --ontology} <ontology
    file> ]
                [ {-u | --db-user} <DB user> ]
                [ {-x | --xml-catalog} <xml
    catalog file> ]

OPTIONS
    -a <fact file>, --facts <fact file>
        User-supplied constant fact file

    -c <constraint file>, --constraint
    <constraint file>
```

```
User-supplied DB constraint file

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--enable-annotations
    enable annotation properties defined
in the ontology. Default:
    false

--facts-base-iri <Base IRI of facts in
fact file>
    The base IRI used for the facts taken
from the fact file.

--facts-format <format of facts file>
    The format of the materialized
ontology. Default: infer from file extension

    This options value is restricted to
the following set of values:
        rdfxml
        turtle
        ntriples
        nquads
        trig
        jsonld

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
    --ontop-views <lenses file>
        User-supplied lenses file. Lenses were
formerly named Ontop views.

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in
```

Ontop native format (.obda)

```

-o <output>, --output <output>
    output file in the CSV format. If not
specified, will print the
    results in the standard output.

-p <properties file>, --properties
<properties file>
    Properties file

-q <queryFile>, --query <queryFile>
    SPARQL SELECT query file

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)

-x <xml catalog file>, --xml-catalog <xml
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports

```

Example 1

Execute a SPARQL query using Ontop mappings.

```

$ ./ontop query -m university-complete.obda \
    -t university-complete.owl \
    -p university-complete.properties \
    -q q1.txt

x
http://www.Department0.University0.edu/GraduateStudent
http://www.Department0.University0.edu/GraduateStudent

```

```
http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent
```

where `q1.txt` contains the SPARQL query, e.g.:

```
PREFIX : <http://example.org/voc#>
SELECT ?x { ?x a :GraduateStudent . }
```

Example 2

Execute a SPARQL query using R2RML mappings and output the query result to a file.

```
$ ./ontop query -m university-complete.ttl \
               -t university-complete.owl \
               -p university-complete.properties \
               -q q1.txt \
               -o q1.csv

$ cat q1.csv
x
http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent
```

ontop extract-db-metadata

Stable since 4.1.0.

This command extracts the metadata from the database and

serializes it into a JSON file. This file can later on be passed as an argument to many other commands.

```
$ ./ontop help extract-db-metadata
NAME
    ontop extract-db-metadata - Extract the DB
    metadata and serialize it
    into an output JSON file

SYNOPSIS
    ontop extract-db-metadata [ {-o | --
    output} <output> ]
    {-p | --properties} <properties
    file>

OPTIONS
    -o <output>, --output <output>
    output file

    -p <properties file>, --properties
    <properties file>
    Properties file
```

Example

```
$ ./ontop extract-db-metadata -p
mobility.properties -o db-metadata.json
```

sh

```
{
  "relations" : [ {
    "uniqueConstraints" : [ {
      "name" : "metadata_pkey",
      "determinants" : [ "\"id\"" ],
      "isPrimaryKey" : true
    } ],
    "foreignKeys" : [ {
      "name" :
      "fk_metadata_station_id_station_pk",
      "from" : {
        "relation" : [ "\"mobility\"",
        "\"metadata\"" ],
        "columns" : [ "\"station_id\"" ]
```

json

```

    },
    "to" : {
        "relation" : [ "\"mobility\"",
        "\"station\"" ],
        "columns" : [ "\"id\"" ]
    }
} ],
"columns" : [ {
    "name" : "\"id\"",
    "isNullable" : false,
    "datatype" : "bigserial"
}, {
    "name" : "\"created_on\"",
    "isNullable" : true,
    "datatype" : "timestamp"
}, {
    "name" : "\"json\"",
    "isNullable" : true,
    "datatype" : "jsonb"
}, {
    "name" : "\"station_id\"",
    "isNullable" : true,
    "datatype" : "int8"
} ],
"name" : [ "\"mobility\"", "\"metadata\"" ]
},
{
    "uniqueConstraints" : [ {
        "name" : "station_pkey",
        "determinants" : [ "\"id\"" ],
        "isPrimaryKey" : true
    }, {
        "name" :
"uc_station_stationcode_stationtype",
        "determinants" : [ "\"stationcode\"",
        "\"stationtype\"" ],
        "isPrimaryKey" : false
    } ],
    "columns" : [ {
        "name" : "\"id\"",
        "isNullable" : false,
        "datatype" : "bigserial"
    }, {
        "name" : "\"name\"",
        "isNullable" : false,
        "datatype" : "varchar(255)"
    }, {

```

```

        "name" : "\"stationtype\"",
        "isNullable" : false,
        "datatype" : "varchar(255)"
    } ],
    "name" : [ "\"mobility\"", "\"station\"" ]
} ],
"metadata" : {
    "dbmsProductName" : "PostgreSQL",
    "dbmsVersion" : "13.1",
    "driverName" : "PostgreSQL JDBC Driver",
    "driverVersion" : "42.2.8",
    "quotationString" : "\"",
    "extractionTime" : "2021-02-25T15:56:09",
    "idFactoryType" : "POSTGRESQL"
}
}

```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Getting started](#)

[Standards compliance →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
 - HTTP API
 - Example
 - Entries
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Predefined query endpoint (beta)

Since 4.1.0.

The predefined query endpoint enables the specification of custom low-latency Web APIs in a declarative manner. It relies on two main components:

1. A JSON-based configuration file, describing the API interface, its parameters and optionally a JSON-LD frame for shaping the returned data.
2. A SPARQL query in charge of retrieving and perhaps transforming the data. The values of the API parameters are plugged as constants into the SPARQL query.

WARNING

At the moment only CONSTRUCT queries are supported.

This endpoint brings one important benefit in terms of performance with respect to the SPARQL endpoint: it is able to reuse reformulations from similar queries. This is particularly valuable for "lookup" requests involving many optional triples, and for which latency is critical.

This endpoint can be deployed using the command `ontop endpoint` or the Docker image `ontop/ontop`.

HTTP API

```
GET /predefined/{queryId}?{param1}={value1}&{param2}={value2}&{param3}={value3}
```

Example

Configuration

```

{
  "queries": {
    "lodging": {
      "queryType": "graph",
      "name": "Lodging businesses",
      "description": "JSON-LD snippets for
lodging businesses, based on schema.org.",
      "return404IfEmpty": true,
      "resultStreaming": false,
      "parameters": {
        "id": {
          "description": "Internal
identifier of the lodging business",
          "type": "xsd:string",
          "safeForRandomGeneration":
true,
          "required": true
        },
        "language": {
          "description": "Language (e.g.
it, de, en, ru). Default: en",
          "type": "xsd:string",
          "safeForRandomGeneration":
false,
          "required": false
        }
      },
      "frame": {
        "@context": "https://schema.org",
        "@type": [
          "LodgingBusiness"
        ]
      }
    }
  }
}

```

Predefined queries

```
[lodging]
query=""
BASE <http://noi.example.org/data/accommodation/>
PREFIX schema: <http://schema.org/>
PREFIX : <http://noi.example.org/ontology/odh#>
CONSTRUCT {
    ?h a schema:LodgingBusiness ; schema:name ?
nameStr ; schema:priceRange ?priceRange .
}
WHERE {
    # Convert incoming parameters
    BIND (IRI(?id) AS ?h)
    BIND (COALESCE(?language, "en") AS ?l)

    ?h a schema:LodgingBusiness .

    OPTIONAL {
        ?h schema:name ?name .
        BIND(str(?name) AS ?nameStr)
        FILTER(lang(?name) = ?l)
    }
    OPTIONAL {
        ?h schema:priceRange ?priceRange .
    }
}
""
```

Cache for the JSON-LD contexts

Optionally, JSON-LD context documents can be stored in a cache and adapted if needed. When they are not cached, these documents are directly fetched from the Web.

```
{
  "https://schema.org": {
    "@context": {
      "@vocab": "http://schema.org/"
    }
  }
}
```

json

Request

```
curl http://localhost:8080/predefined/lodging?
id=32E7BE648E7B11D181AB006097B896BA&language=de
```

Results:

```
{
  "@context": "https://schema.org",
  "@id":
"http://noi.example.org/data/accommodation/32E7BE648E7

  "@type": "LodgingBusiness",
  "name": "Alpenblick - Gasthaus"
}
```

Entries

Top-level key

Key	Type	Description
<code>queries</code>	Map of predefined queries	Groups predefined queries by id.

Query keys

Key	Type	Description
<code>queryType</code>	String	At the moment, only <code>graph</code> is supported. Corresponds to CONSTRUCT queries.
<code>name</code>	String	Name of the predefined query.
<code>description</code>	String	Description of the predefined query.
		If true, returns a

<code>return404IfEmpty</code>	Boolean	404 status code when the result set is empty.
<code>resultStreaming</code>	Boolean	If true, uses the chunked transfer encoding.
<code>parameters</code>	Map of parameters	Groups parameters by id.
<code>frame</code>	JSON object	<i>JSON-LD frame</i>) for shaping the result set when the JSON-LD format is selected.

Parameter keys

Key	Type	Description
<code>description</code>	String	Description of the parameter.
<code>type</code>	String	Type of the parameter. Can be <code>iri</code> for an IRI, any XSD datatype like <code>xsd:string</code> with the prefix <code>xsd:</code> or the IRI string of an RDF datatype.
		If true, replacing the parameter value by a random value for generating a "reference" reformulated query can be considered as safe. Replacing afterwards

<code>safeForRandomGeneration</code>	Boolean	the random value but the real parameter value in the reference reformulated query should produce the expected result. Usually, identifiers are safe while languages are not because the columns to consider differ from a language to another.
<code>required</code>	Boolean	If true, a value must be given for this parameter in the HTTP request.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Ontop Mapping Language](#)

[Query logging →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

PostgreSQL

Through the [PostgreSQL](#) connector, Ontop is able to construct VKGs on external Postgres databases.

Limitations & Exceptions

- Accessing JSON object fields with the "arrow operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to PostgreSQL:

```
jdbc.url =
jdbc:postgresql://localhost:5432/database
jdbc.user = admin
jdbc.password = password
jdbc.driver = org.postgresql.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the PostgreSQL array type `T[]`. When used with the [flatten lens](#), it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the `JSON` and `JSONB` datatypes, which are both recognized by Ontop. For these types, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In PostgreSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
```

```
JSON_EXTRACT_PATH(TO_JSON(my_struct), '$.my_attribute')
```

WARNING

The "arrow operators" used to access JSON objects in PostgreSQL are not supported in Ontop. Please use JSON functions instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← [Oracle](#)

[Presto](#) →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
 - Limitations & Exceptions
 - Database Connection

MySQL

Through the [MySQL](#) connector, Ontop is able to construct VKGs on MySQL databases.

Limitations & Exceptions

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.
- Accessing JSON object fields with the "dot operator" is not supported (see below).
- MySQL 5.x comes with some limitations (e.g. no anonymous bnode support, no regexp replace, no nested data support). MySQL 5.x reached its end-of-life, so please consider upgrading.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to MySQL:

```
sh
jdbc.url =
jdbc:mysql://localhost:3306/defaultdatabase?
useCursorFetch=true
jdbc.user = user
jdbc.password = password
jdbc.driver = com.mysql.cj.jdbc.Driver
```

WARNING

The parameter `useCursorFetch=true` needs to be added to the JDBC URL for streaming to be enabled. Otherwise, the JDBC driver will block until the full result set is fetched, which may cause out-of-memory

exceptions when materializing or running large queries.

Nested Type Support

Ontop implements explicit compatibility with the MySQL type `JSON` which can be used with the [flatten lens](#). However, it cannot infer the type of the flattened result column.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [MySQL's JSON functions](#) can be used to further work with them.

WARNING

Accessing object fields using the "arrow operator" is not supported.

WARNING

Nested type only supported for MySQL ≥ 8.0

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MS SQL Server](#)

[Oracle →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- MS SQL Server

MariaDB

Supported since 5.0.0.

Through the [MariaDB](#) connector, Ontop is able to construct VKGs on MariaDB databases.

Limitations & Exceptions

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to MariaDB:

```
jdbc.url =  
jdbc:mariadb://localhost:3306/defaultdatabase  
jdbc.user = user  
jdbc.password = password  
jdbc.driver = org.mariadb.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the MariaDB type `JSON` which can be used with the [flatten lens](#). However, it cannot infer the type of the flattened result column.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [MariaDB's JSON functions](#) can be used to further work with them.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← H2](#)

[MS SQL Server →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB

MS SQL Server

- Limitations & Exceptions
- Database Connection
- Nested Type Support

MS SQL Server

Through the [MS SQL Server](#) connector, Ontop is able to construct VKGs on SQL Server databases.

Limitations & Exceptions

- RegEx SPARQL functions are not supported.
- The `position` argument for the [Flatten Lens](#) cannot be used with SQLServer.
- Support for `DATETRUNC` in SQL Server is very recent (introduced in SQL Server 2022)

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to SQLServer:

```
sh
jdbc.url =
jdbc:sqlserver://localhost:1433;databaseName=defaultda

jdbc.user = user
jdbc.password = password
jdbc.driver =
com.microsoft.sqlserver.jdbc.SQLServerDriver
```

Nested Type Support

Nested data types are not supported by SQLServer. However, Ontop allows the usage of the [flatten lens](#) over string columns that contain JSON-encoded arrays. The output type of the flattened result column cannot be inferred.

WARNING

The `position` argument for flatten lenses cannot be used with SQLServer.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [SQLServer's JSON functions](#) can be used to further work with them.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MariaDB](#)

[MySQL →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
 - Limitations & Exceptions

Oracle

Through the [Oracle](#) connector, Ontop is able to construct VKGs on external Oracle databases.

Limitations & Exceptions

- Ontop cannot perform inference based on nested data types in Oracle.
- Ontop represents the "big integer" datatype as `NUMBER(19)`.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Oracle:

```
sh
jdbc.url =
jdbc:oracle:thin:@//localhost:49161/defaultcontainer

jdbc.user = user
jdbc.password = password
jdbc.driver = oracle.jdbc.OracleDriver
```

Nested Type Support

Ontop does not implement explicit compatibility with nested datatypes in Oracle. However, Ontop allows the usage of the [flatten lens](#) over string columns that contain JSON-encoded arrays. The output type of the flattened result column cannot be inferred.

To use the flatten lens with Oracle array types, they must first be converted to JSON strings.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [Oracle's JSON functions](#) can be used to further work with them.

WARNING

Accessing object fields using the "dot operator" is not supported. Please use SQL JSON functions instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MySQL](#)

[PostgreSQL →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

DB2

Through the IBM DB2 connector, Ontop is able to construct VKGs on DB2 databases.

Limitations & Exceptions

- SPARQL *UUID* and *hash* functions are not supported for DB2.
- Nested data types and the *flatten lens* are not supported for DB2

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DB2:

```
sh
jdbc.url =
jdbc:db2://localhost:50000/defaultdatabase
jdbc.user = user
jdbc.password = password
jdbc.driver = com.ibm.db2.jcc.DB2Driver
```

Nested Type Support

Nested data types and the *flatten lens* are not supported for DB2.

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Snowflake

Supported since 5.0.0.

Through the [Snowflake](#) connector, Ontop is able to construct VKGs on cloud-based Snowflake databases.

Limitations & Exceptions

WARNING

Integrity constraints are often missing in Snowflake. Make sure to specify the missing ones in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

If you want to use the Snowflake JDBC Driver with Java 17, the following option must be added to the JVM `--add-opens java.base/java.nio=ALL-UNNAMED`.

Alternatively, you can add the parameter `JDBC_QUERY_RESULT_FORMAT=JSON` to the JDBC URL to avoid using Apache Arrow.

- Due to an issue in the JDBC, quotation marks cannot always be used when defining an alias. Because of this, it is recommended to choose column names without special characters where possible.
- Accessing struct fields with the "colon operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Snowflake:

```
sh
jdbc.url =
jdbc:snowflake://dummyuser.snowflakecomputing.com/?
warehouse=dummywarehouse&db=dummydb&schema=dummyschema

jdbc.user = user
jdbc.password = password
jdbc.driver =
net.snowflake.client.jdbc.SnowflakeDriver
```

Nested Type Support

Ontop implements explicit compatibility with the `ARRAY` type, which can be used with the `flatten lens`. However, it is not able to automatically infer the type of the resulting column.

WARNING

Due to an issue mentioned above, using special characters inside the `output` and `position` fields of the flatten lens for Snowflake will fail. Furthermore, these fields will always be case-insensitive.

Struct Access

Individual object fields can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_object['my_field'] FROM ...
```

WARNING

Accessing object fields using the "colon operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Presto](#)

[SparkSQL →](#)

Guide

- Introduction
- Key concepts
 - Virtual Knowledge Graph (VKG)
 - RDF
 - SPARQL query
 - Mappings
 - Ontology
 - VKG specification
 - SPARQL endpoint
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Key concepts

Virtual Knowledge Graph (VKG)

A Knowledge Graph (KG) is, in our terminology, a graph using the RDF data model.

A Virtual KG (VKG) is a virtual representation in RDF of non-RDF data, which is generally relational data. With a VKG, the data remains in the data sources in its original format but can be virtually represented as an RDF graph.

RDF

The [Resource Description Framework \(RDF\)](#) is one of the two main data models for graphs (together with [property graphs](#)). RDF mainly targets data integration applications while property graphs are used for building graph databases.

In RDF, data is modelled using classes and properties.

Starting from 3.0, Ontop supports [RDF 1.1](#) .

SPARQL query

[SPARQL](#) is the standard query language for RDF graphs.

Ontop is capable of answering SPARQL queries expressed over the VKG. Ontop translates these SPARQL queries into SQL queries, which are then executed over the relational data sources.

Ontop supports a large fragment of [SPARQL 1.1](#) .

Mappings

Mappings specify the correspondence between the data models of the relational data sources and the RDF graph. Ontop supports the [R2RML standard mapping language](#) and the [Ontop mapping language](#), which is fully interoperable with R2RML.

Ontology

An ontology specifies the formal relations between the classes and properties used by the RDF graph. It is mainly used for enriching the RDF graph by, for instance, taking account of class hierarchies.

Ontop supports lightweight ontologies expressed in [RDFS](#) or in the slightly more expressive [OWL 2 QL](#) fragment of OWL.

VKG specification

VKG specifications are composed of mappings and optionally of ontologies.

SPARQL endpoint

A [SPARQL endpoint](#) is a standardized HTTP-based Web API. It makes the RDF graph queryable by any HTTP client.

Ontop enables VKG specifications to be deployed as SPARQL endpoints.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Introduction](#)

[Getting started →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Tutorial
- Using Ontop
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Getting started

Tutorial

If you are new to Ontop and Virtual Knowledge Graphs, we encourage you to have a look [at the official tutorial](#).

Using Ontop

Ontop is distributed under various forms. They can be downloaded on [Github](#) , [Docker Hub](#) , [Sourceforge](#) and in the Protégé plugin repository.

Mapping designer

For editing and testing your mappings, you can use our plugin of the [Protégé ontology editor](#) . You can download the latest stable release directly from Protégé. Alternatively, pre-releases can be found on [Github](#) and [Sourceforge](#) .

Limited support of advanced features

In particular, the Protégé plugin doesn't support lenses. For advanced usage, we recommend using text editors or considering commercial offerings.

Deployment

Once your mappings and your ontology are ready, you can deploy your VKG as a SPARQL endpoint. The Ontop endpoint is available both as a [CLI command](#) (`ontop endpoint`) and as a [Docker image](#) .

You can also use the Ontop endpoint during development as it

embeds [a nice YASGUI client](#) and an optional portal (since 4.0-beta-1) containing pre-defined queries.

Command Line Interface

Want to materialize your VKG, convert your mappings into R2RML, bootstrap your mappings or start a SPARQL endpoint? You can use [the CLI](#) for that. It can be found on [Github](#) and [Sourceforge](#).

Former solutions

Historically, Ontop has been made available under other means that we don't recommend anymore.

Java API (not recommended)

It remains possible to use Ontop as a Java API through [RDF4J](#) although we recommend the HTTP SPARQL endpoint as a first option (think of a microservice). Why? Because such a Java API would add many dependencies to your project and constraint you to use certain versions of Java.

Ontop-rdf4j is available on [Maven Central](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Key concepts](#)

[Command Line Interface →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
 - Setup Ontop CLI
 - ontop endpoint
 - ontop materialize
 - ontop mapping
 - ontop bootstrap
 - ontop query
 - ontop extract-db-metadata
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Command Line Interface

Ontop ships a shell script (`ontop` for *nix) and a bat file (`ontop.bat` for Windows) exposing the core functionality and several utilities through the command line interface. It is an easy way to get the system quickly set-up, test for correct execution, and query or materialize as needed.

- Setup
- ontop endpoint
- ontop materialize
- ontop mapping
- ontop bootstrap
- ontop query
- ontop extract-db-metadata

Setup Ontop CLI

First, you have to download Ontop latest CLI zip from our download pages ([Github](#) or [Sourceforge](#)). Unzip it in a folder. Open the command line terminal and cd to that folder. For Windows use the `ontop.bat` file, for Linux and OS X use the `ontop` file.

```
$ ./ontop help
usage: ontop <command> [ <args> ]

Commands are:
  --version          Show version of ontop
  bootstrap          Bootstrap ontology and
mapping from the database
  endpoint            Start a SPARQL endpoint
powered by Ontop
  extract-db-metadata Extract the DB metadata
and serialize it into an output JSON file
  help               Display help information
```

<code>materialize</code>	Materialize the RDF graph exposed by the mapping and the OWL ontology
<code>query</code>	Query the RDF graph exposed by the mapping and the OWL ontology
<code>validate</code>	Validate Ontology and Mappings
<code>mapping</code>	Manipulate mapping files

JDBC configuration

JDBC drivers are software implemented by third parties (often the same developers of the database system) that handle interaction with the database in their own proprietary protocols. You will need to manually download the JDBC drivers for your database management system (e.g., [PostgreSQL JDBC drivers](#)) and put them into the `jdbc` directory.

PATH

Consider putting the directory of ontop to your `PATH` .

Logging level

It can be set using the environment variable `ONTOP_LOG_LEVEL` . Set it to `DEBUG` to get most details, including generated SQL queries. Alternatively, you can change the value of the `level` attribute of the `root` element at the end of the file `log/logback.xml` .

Property file

Most commands below require or accept as input a property file. This is where you will specify the JDBC connection parameters. A basic property file template can be found [here](#).

`ontop endpoint`

`ontop endpoint` deploys a SPARQL endpoint locally at the

address `/sparql` and by default on the port 8080. It powers [our official Docker](#), so feel free to use the Docker image instead of the CLI command if it is more convenient for you.

It offers several advanced options:

- *Lazy initialization*: Ontop offline tasks (such as DB metadata extraction and mapping processing) are triggered after receiving the first SPARQL query. This is useful when using a Docker-Compose with Ontop and a DB image that needs to be initialized first.
- *Development mode (since 4.0-beta-1)*: restarts the endpoint every time the configuration files are changed. It also exposes a GET/POST method `/ontop/reformulate` accepting a SPARQL query as param as any SPARQL endpoint but returning the reformulated SQL query as result.
- *Portal (since 4.0-beta-1)*: Includes groups of pre-defined SPARQL queries into the welcome page. See the following [example of portal file](#) in the TOML format.
- *Predefined query endpoint (since 4.1.0)*
- *Ontology made downloadable (since 4.2.0)*.

```
$ ./ontop help endpoint
NAME
    ontop endpoint - Start a SPARQL endpoint
    powered by Ontop

SYNOPSIS
    ontop endpoint [ {-a | --facts} <fact
    file> ]
                    [ {-c | --constraint} <constraint
    file> ]
                    [ --contexts <JSON-LD context file
    for predefined queries> ]
                    [ --cors-allowed-origins <origins>
    ]
                    [ {-d | --db-metadata} <db-
    metadata file> ]
                    [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                    [ --db-url <DB URL> ] [ --dev ] [
    --disable-portal-page ]
                    [ --enable-annotations ] [ --
```

```
enable-download-ontology ]
    [ --facts-base-iri <Base IRI of
facts in fact file> ]
    [ --facts-format <format of facts
file> ]
    [ {-l | --lenses | -v | --ontop-
views} <lenses file> ]
    [ --lazy ] {-m | --mapping}
<mapping file>
    [ {-p | --properties} <properties
file> ] [ --port <port> ]
    [ --portal <endpoint portal file>
]
    [ --predefined-config <predefined
query JSON config file> ]
    [ --predefined-queries <predefined
query TOML file> ]
    [ --sparql-rules <SPARQL rules
file> ]
    [ {-t | --ontology} <ontology
file> ]
    [ {-u | --db-user} <DB user> ]
    [ {-x | --xml-catalog} <xml
catalog file> ]
```

OPTIONS

```
-a <fact file>, --facts <fact file>
    User-supplied constant fact file

-c <constraint file>, --constraint
<constraint file>
    User-supplied DB constraint file

--contexts <JSON-LD context file for
predefined queries>
    File containing JSON-LD contexts for
predefined queries

--cors-allowed-origins <origins>
    CORS allowed origins

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)
```



```

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--dev
    development mode

--disable-portal-page
    Disable the portal page (/index.html)
    of the SPARQL endpoint.

--enable-annotations
    enable annotation properties defined
    in the ontology. Default:
    false

--enable-download-ontology
    Allow to download the ontology as a
    plain text file (/ontology).
    Default: false

--facts-base-iri <Base IRI of facts in
fact file>
    The base IRI used for the facts taken
    from the fact file.

--facts-format <format of facts file>
    The format of the materialized
    ontology. Default: infer from file extension

    This options value is restricted to
    the following set of values:
        rdfxml
        turtle
        ntriples
        nquads
        trig
        jsonld

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
--ontop-views <lenses file>
    User-supplied lenses file. Lenses were
    formerly named Ontop views.

```

```
--lazy
    lazy initialization

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in
    Ontop native format (.obda)

-p <properties file>, --properties
<properties file>
    Properties file

--port <port>
    port of the SPARQL endpoint

--portal <endpoint portal file>
    endpoint portal file (including title
    and queries)

--predefined-config <predefined query JSON
config file>
    predefined query config file

    This option is required if any of the
    following options are
    specified: --predefined-queries

--predefined-queries <predefined query
TOML file>
    predefined SPARQL queries file

    This option is required if any of the
    following options are
    specified: --predefined-config

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)
```

```
-x <xml catalog file>, --xml-catalog <xml
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports
```

Example

```
$ ./ontop endpoint -m university-complete.obda \
    -t university-complete.ttl \
    -p university-
complete.properties \
    --cors-allowed-origins=*
```

ontop materialize

This command provides a "materialization utility". Materialization is helpful when you want to generate RDF data out of your database, using the provided mappings. This utility will take all the triples that the mapping can produce from the data source, and write them to the output. `ontop materialize` does not need any query file, but instead, needs the user to specify a format in which he/she wants the output (either to terminal or output file). The user can choose between three output formats: [Turtle](#) , [N-triples](#) or [RDF/XML](#) . For very large datasets, producing the output might take some time.

The compression option has been added in 5.2.0.

```
$ ./ontop help materialize
NAME
    ontop materialize - Materialize the RDF
graph exposed by the mapping
    and the OWL ontology

SYNOPSIS
    ontop materialize [ {-a | --facts} <fact
file> ]
                        [ {-c | --constraint} <constraint
```

```
file> ]
        [ --compression <output
compression> ]
        [ {-d | --db-metadata} <db-
metadata file> ]
        [ --db-driver <DB driver> ] [ --
db-password <DB password> ]
        [ --db-url <DB URL> ] [ --enable-
annotations ]
        [ {-f | --format} <output format>
]
        [ --facts-base-iri <base IRI of
facts in fact file> ]
        [ --facts-format <format of facts
file> ]
        [ {-l | --lenses | -v | --ontop-
views} <Lenses file> ]
        {-m | --mapping} <mapping file> [
--no-streaming ]
        [ {-o | --output} <output> ]
        [ {-p | --properties} <properties
file> ] [ --separate-files ]
        [ --sparql-rules <SPARQL rules
file> ]
        [ {-t | --ontology} <ontology
file> ]
        [ {-u | --db-user} <DB user> ]
        [ {-x | --xml-catalog} <xml
catalog file> ]
```

OPTIONS

-a <fact file>, --facts <fact file>
RDF fact file

-c <constraint file>, --constraint
<constraint file>
User-supplied DB constraint file

--compression <output compression>
The compression format of the
materialized RDF graph. Default: no
compression

This options value is restricted to
the following set of values:

- gzip
- zip

```

no_compression

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--enable-annotations
    enable annotation properties defined
in the ontology. Default:
    false

-f <output format>, --format <output
format>
    The format of the materialized RDF
graph. Default: rdxml

    This options value is restricted to
the following set of values:
        rdxml
        turtle
        ntriples
        nquads
        trig
        jsonld

--facts-base-iri <base IRI of facts in
fact file>
    The base IRI of facts in the fact file
to resolve relative IRIs. If
    not provided, a random IRI is
generated.

--facts-format <format of facts file>
    The format of the 'facts' input file.

    This options value is restricted to
the following set of values:
        rdxml

```

```

turtle
ntriples
nquads
trig
jsonld

-l <Lenses file>, --lenses <Lenses file>,
-v <Lenses file>,
--ontop-views <Lenses file>
    User-supplied lenses file. Lenses were
    formerly named Ontop views.

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in
    Ontop native format (.obda)

--no-streaming
    All the SQL results of one big query
    will be stored in memory. Not
    recommended. Default: false.

-o <output>, --output <output>
    output file (default) or prefix (only
    for --separate-files)

-p <properties file>, --properties
<properties file>
    Properties file

--separate-files
    generating separate files for
    different classes/properties. This is
    useful for materializing large OBDA
    setting. Default: false.

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)

-x <xml catalog file>, --xml-catalog <xml

```

```
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports
```

Examples

```
$ ./ontop materialize -m university-complete.obda
\
    -t university-complete.ttl \
    -p university-
complete.properties \
    -f turtle \
    -o materialized-triples.ttl
```

In case you have some lenses:

```
$ ./ontop materialize -m mapping.ttl \
    -t ontology.ttl \
    -l lenses.json \
    -p configuration.properties
\
    -f turtle \
    -o materialized-triples.ttl
```

ontop mapping

This command collects several useful sub-commands for dealing with mappings files.

```
$ ./ontop help mapping
NAME
    ontop mapping - Manipulate mapping files

SYNOPSIS
    ontop mapping { pretty-r2rml | to-obda |
to-r2rml | v1-to-v3 } [--]
    [cmd-options]
```

```

Where command-specific options [cmd-
options] are:
    pretty-r2rml: {-i | --input}
<input.ttl> {-o | --output}
               <pretty.ttl>
    to-obda: {-i | --input} <mapping.ttl>
[ {-o | --output} <mapping.obda> ]
    to-r2rml: [ {-l | --lenses | -v | --
ontop-views} <lenses file> ] [ {-t | --ontology}
<ontology.owl> ]
               {-i | --input} <mapping.obda>
[ {-o | --output} <mapping.ttl> ]
               [ {-p | --properties}
<properties file> ] [ {-d | --db-metadata} <db-
metadata file> ]
               [ --force ]
    v1-to-v3: [ --simplify-projection ] {-
m | --mapping} <mapping file>
               [ --overwrite ] [ {-o | --
output} <mapping.obda> ]

See 'ontop help mapping <command>' for
more information on a specific command.

```

ontop mapping to-r2rml

Supports automatically converting mappings from Ontop native format (`.obda`) to the `R2RML` standard format. Since 4.1.0, by default, it expects DB credentials (for extracting the DB metadata) or a DB metadata file. This requirement can be bypassed using the option `--force`.

```

$ ./ontop help mapping to-r2rml
NAME
    ontop mapping to-r2rml - Convert ontop
native mapping format (.obda) to
R2RML format

SYNOPSIS
    ontop mapping to-r2rml [ {-d | --db-
metadata} <db-metadata file> ]
                          [ --force ] {-i | --input}
<mapping.obda>
                          [ {-l | --lenses | -v | --ontop-

```



```
views} <lenses file> ]
    [ {-o | --output} <mapping.ttl> ]
    [ {-p | --properties} <properties
file> ]
    [ {-t | --ontology} <ontology.owl>
]
```

OPTIONS

```
-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--force
    Force the conversion in the absence of
DB metadata

-i <mapping.obda>, --input <mapping.obda>
    Input mapping file in Ontop native
format (.obda)

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
--ontop-views <lenses file>
    User-supplied lenses file. Lenses were
formerly named Ontop views.

-o <mapping.ttl>, --output <mapping.ttl>
    Output mapping file in R2RML format
(.ttl)

-p <properties file>, --properties
<properties file>
    Properties file

-t <ontology.owl>, --ontology
<ontology.owl>
    OWL ontology file
```

ontop mapping to-obda

Supports automatically converting mappings from [R2RML](#) standard format to Ontop native format ([.obda](#)):

```
$ ./ontop help mapping to-obda
```

```
NAME
    ontop mapping to-obda - Convert R2RML
    format to ontop native mapping
    format (.obda)

SYNOPSIS
    ontop mapping to-obda {-i | --input}
    <mapping.ttl>
                        [ {-o | --output} <mapping.obda> ]

OPTIONS
    -i <mapping.ttl>, --input <mapping.ttl>
        Input mapping file in R2RML format
        (.ttl)

    -o <mapping.obda>, --output <mapping.obda>
        Output mapping file in Ontop native
        format (.obda)
```

`ontop mapping pretty-r2rml`

Provides automatic formatting and prettifying facilities for mappings files:

```
$ ./ontop help mapping pretty-r2rml
NAME
    ontop mapping pretty-r2rml - prettify
    R2RML file using Jena

SYNOPSIS
    ontop mapping pretty-r2rml {-i | --input}
    <input.ttl>
                        {-o | --output} <pretty.ttl>

OPTIONS
    -i <input.ttl>, --input <input.ttl>
        Input mapping file in the turtle R2RML
        format (.ttl)

    -o <pretty.ttl>, --output <pretty.ttl>
        Output mapping file in the turtle
        R2RML format (.ttl)
```

ontop bootstrap

This command allows the automatic generation of mappings and ontology starting from a database schema. The generated output can be used as-is or further customized manually (e.g., to used different ontological modeling choices and corresponding mappings). In both cases, it helps substantially reducing the user effort involved in setting up the ontology and mappings of a VKG specification.

```
$ ./ontop help bootstrap
NAME
    ontop bootstrap - Bootstrap ontology and
    mapping from the database

SYNOPSIS
    ontop bootstrap {-b | --base-iri} <base
    IRI>
                    [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                    [ --db-url <DB URL> ] {-m | --
    mapping} <mapping file>
                    [ {-p | --properties} <properties
    file> ]
                    {-t | --ontology} <ontology file>
                    [ {-u | --db-user} <DB user> ]

OPTIONS
    -b <base IRI>, --base-iri <base IRI>
        Base IRI of the generated mapping

    --db-driver <DB driver>
        DB driver (overrides the properties)

    --db-password <DB password>
        DB password (overrides the properties)

    --db-url <DB URL>
        DB URL (overrides the properties)

    -m <mapping file>, --mapping <mapping
    file>
        Output mapping file in the Ontop
        native format (.obda)
```

```
-p <properties file>, --properties
<properties file>
    Properties file

-t <ontology file>, --ontology <ontology
file>
    Output OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)
```

ontop query

The `ontop query` command is designed for helping users to test their system quickly using the command line utilities. You can use this command if you already have a scenario test case including:

- the ontology (RDFS or OWL) and the mappings (obda or R2RML) files,
- a working database to connect to,
- a SPARQL query file

The `ontop query` command helps you to set up the system, runs the query from the query string file over it, and gets the results either in output file or terminal output. What the script actually does is to set up Ontop using the ontology and the mapping files, parse the query from the file and execute it over the created instance of Ontop.

Note that `ontop query` is NOT intended to be used in production and for benchmarking purposes. Most of its execution time is dedicated to offline tasks like DB metadata extraction and mapping processing. Query answering (i.e. answering the SPARQL query) takes usually much less time. For production and benchmarking purposes, please consider [deploying Ontop as a SPARQL endpoint](#).

The results are turned in the CSV format.

WARNING

At the moment only SELECT queries are supported by this command. See #222 .

```
$ ./ontop help query
NAME
    ontop query - Query the RDF graph exposed
    by the mapping and the OWL
    ontology

SYNOPSIS
    ontop query [ {-a | --facts} <fact file> ]
                [ {-c | --constraint} <constraint
    file> ]
                [ {-d | --db-metadata} <db-
    metadata file> ]
                [ --db-driver <DB driver> ] [ --
    db-password <DB password> ]
                [ --db-url <DB URL> ] [ --enable-
    annotations ]
                [ --facts-base-iri <Base IRI of
    facts in fact file> ]
                [ --facts-format <format of facts
    file> ]
                [ {-l | --lenses | -v | --ontop-
    views} <lenses file> ]
                {-m | --mapping} <mapping file> [
    {-o | --output} <output> ]
                [ {-p | --properties} <properties
    file> ]
                {-q | --query} <queryFile>
                [ --sparql-rules <SPARQL rules
    file> ]
                [ {-t | --ontology} <ontology
    file> ]
                [ {-u | --db-user} <DB user> ]
                [ {-x | --xml-catalog} <xml
    catalog file> ]

OPTIONS
    -a <fact file>, --facts <fact file>
        User-supplied constant fact file

    -c <constraint file>, --constraint
    <constraint file>
```

```

User-supplied DB constraint file

-d <db-metadata file>, --db-metadata <db-
metadata file>
    User-supplied db-metadata file

--db-driver <DB driver>
    DB driver (overrides the properties)

--db-password <DB password>
    DB password (overrides the properties)

--db-url <DB URL>
    DB URL (overrides the properties)

--enable-annotations
    enable annotation properties defined
in the ontology. Default:
    false

--facts-base-iri <Base IRI of facts in
fact file>
    The base IRI used for the facts taken
from the fact file.

--facts-format <format of facts file>
    The format of the materialized
ontology. Default: infer from file extension

    This options value is restricted to
the following set of values:
        rdfxml
        turtle
        ntriples
        nquads
        trig
        jsonld

-l <lenses file>, --lenses <lenses file>,
-v <lenses file>,
    --ontop-views <lenses file>
        User-supplied lenses file. Lenses were
formerly named Ontop views.

-m <mapping file>, --mapping <mapping
file>
    Mapping file in R2RML (.ttl) or in

```

Ontop native format (.obda)

```

-o <output>, --output <output>
    output file in the CSV format. If not
specified, will print the
    results in the standard output.

-p <properties file>, --properties
<properties file>
    Properties file

-q <queryFile>, --query <queryFile>
    SPARQL SELECT query file

--sparql-rules <SPARQL rules file>
    User-supplied SPARQL rules file

-t <ontology file>, --ontology <ontology
file>
    OWL ontology file

-u <DB user>, --db-user <DB user>
    DB user (overrides the properties)

-x <xml catalog file>, --xml-catalog <xml
catalog file>
    XML Catalog file (e.g. catalog-
v001.xml generated by Protege) for
    redirecting ontologies imported by
owl:imports

```

Example 1

Execute a SPARQL query using Ontop mappings.

```

$ ./ontop query -m university-complete.obda \
    -t university-complete.owl \
    -p university-complete.properties \
    -q q1.txt

x
http://www.Department0.University0.edu/GraduateStudent
http://www.Department0.University0.edu/GraduateStudent

```

```
http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent
```

where `q1.txt` contains the SPARQL query, e.g.:

```
PREFIX : <http://example.org/voc#>
SELECT ?x { ?x a :GraduateStudent . }
```

Example 2

Execute a SPARQL query using R2RML mappings and output the query result to a file.

```
$ ./ontop query -m university-complete.ttl \
               -t university-complete.owl \
               -p university-complete.properties \
               -q q1.txt \
               -o q1.csv

$ cat q1.csv
x
http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent

http://www.Department0.University0.edu/GraduateStudent
```

ontop extract-db-metadata

Stable since 4.1.0.

This command extracts the metadata from the database and

serializes it into a JSON file. This file can later on be passed as an argument to many other commands.

```
$ ./ontop help extract-db-metadata
NAME
    ontop extract-db-metadata - Extract the DB
    metadata and serialize it
    into an output JSON file

SYNOPSIS
    ontop extract-db-metadata [ {-o | --
    output} <output> ]
    {-p | --properties} <properties
    file>

OPTIONS
    -o <output>, --output <output>
    output file

    -p <properties file>, --properties
    <properties file>
    Properties file
```

Example

```
$ ./ontop extract-db-metadata -p
mobility.properties -o db-metadata.json
```

```
{
  "relations" : [ {
    "uniqueConstraints" : [ {
      "name" : "metadata_pkey",
      "determinants" : [ "\"id\"" ],
      "isPrimaryKey" : true
    } ],
    "foreignKeys" : [ {
      "name" :
      "fk_metadata_station_id_station_pk",
      "from" : {
        "relation" : [ "\"mobility\"",
        "\"metadata\"" ],
        "columns" : [ "\"station_id\"" ]
```

```

    },
    "to" : {
        "relation" : [ "\"mobility\"",
        "\"station\"" ],
        "columns" : [ "\"id\"" ]
    }
} ],
"columns" : [ {
    "name" : "\"id\"",
    "isNullable" : false,
    "datatype" : "bigserial"
}, {
    "name" : "\"created_on\"",
    "isNullable" : true,
    "datatype" : "timestamp"
}, {
    "name" : "\"json\"",
    "isNullable" : true,
    "datatype" : "jsonb"
}, {
    "name" : "\"station_id\"",
    "isNullable" : true,
    "datatype" : "int8"
} ],
"name" : [ "\"mobility\"", "\"metadata\"" ]
},
{
    "uniqueConstraints" : [ {
        "name" : "station_pkey",
        "determinants" : [ "\"id\"" ],
        "isPrimaryKey" : true
    }, {
        "name" :
"uc_station_stationcode_stationtype",
        "determinants" : [ "\"stationcode\"",
        "\"stationtype\"" ],
        "isPrimaryKey" : false
    } ],
    "columns" : [ {
        "name" : "\"id\"",
        "isNullable" : false,
        "datatype" : "bigserial"
    }, {
        "name" : "\"name\"",
        "isNullable" : false,
        "datatype" : "varchar(255)"
    }, {

```

```

        "name" : "\"stationtype\"",
        "isNullable" : false,
        "datatype" : "varchar(255)"
    } ],
    "name" : [ "\"mobility\"", "\"station\"" ]
} ],
"metadata" : {
    "dbmsProductName" : "PostgreSQL",
    "dbmsVersion" : "13.1",
    "driverName" : "PostgreSQL JDBC Driver",
    "driverVersion" : "42.2.8",
    "quotationString" : "\"",
    "extractionTime" : "2021-02-25T15:56:09",
    "idFactoryType" : "POSTGRESQL"
}
}

```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Getting started](#)

[Standards compliance →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance
 - SPARQL 1.1
 - GeoSPARQL 1.0
 - R2RML
 - RDF 1.1
 - OWL 2 QL
 - RDFS
 - Time functions
 - Other functions

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Standards compliance

SPARQL 1.1

Updated for 5.4.0.

In the following table we present a summary of the compliance of the latest version of Ontop with SPARQL 1.1 , where rows correspond to sections of the W3C recommendation and unsupported features are ~~crossed out~~. Most of the features are supported, but some are unsupported or only partially supported.

Section in SPARQL 1.1	Features	Coverag
5. Graph Patterns	BGP , FILTER	2/2
6. Including Optional Values	OPTIONAL	1/1
7. Matching Alternatives	UNION	1/1
8. Negation	MINUS , FILTER [NOT] EXISTS	2/2
9. Property Paths	PredicatePath , InversePath , SequencePath , AlternativePath , NegatedPropertySet , ZeroOrMorePath , OneOrMorePath , ZeroOrOnePath	5/8
10. Assignment	BIND , VALUES	2/2
11. Aggregates	COUNT , SUM , MIN , MAX , AVG , GROUP_CONCAT , SAMPLE	6/6
12. Subqueries	Subqueries	1/1

13. RDF Dataset	GRAPH [NAMED] FROM	2/2
14. Basic Federated Query	SERVICE	0
15. Solution Seqs. & Mods.	ORDER BY DISTINCT OFFSET SELECT REDUCED LIMIT	6/6
16. Query Forms	SELECT ASK CONSTRUCT DESCRIBE	4/4
17.4.1. Functional Forms	BOUND COALESCE NOT EXISTS && IN IF EXISTS = sameTerm NOT IN	11/11
17.4.2. Functions on RDF Terms	isIRI isBlank isLiteral isNumeric lang IRI STRDT UUID str datatype BNode STRLANG STRUUID	11/13
17.4.3. Functions on Strings	STRLEN UCASE STRSTARTS STREND STRBEFORE STRAFTER ENCODE_FOR_URI CONCAT langMatches REGEX SUBSTR LCASE CONTAINS REPLACE	14/14
17.4.4. Functions on Numerics	abs ceil round floor RAND	5/5
17.4.5. Functions on Dates&Times	now month hours seconds tz year day minutes timezone	8/9
17.4.6. Hash Functions	MD5 SHA256 SHA512 SHA1 SHA384	5/5
	xsd:string	

17.5 XPath Constructor Functions	xsd:float xsd:double xsd:decimal xsd:integer xsd:boolean xsd:dateTime	7/7
17.6 Extensible Value Testing	~~user-defined functions~~	0

Limitations

- The 5 hash functions and functions **REPLACE** and **REGEX** for regular expressions have limited support because they heavily depend on the DBMS: not all DBMSs provide all hash functions, and many DBMSs have their own regex dialects. Currently, the SPARQL regular expressions of **REPLACE** and **REGEX** are simply sent to the DBMS by default.
- In the implementation of function **langMatches**, the second argument has to be a constant: allowing variables will have a negative impact on the performance in our framework.
- Cast to **xsd:boolean**. Casting NaN to boolean should result in **"false"^^xsd:boolean**, however this check is not feasible for many DB dialects, hence there is mostly no special handling of these values with support limited to PostgreSQL, Oracle and Spark.
- Cast to **xsd:decimal**. Casting to DECIMAL results in values with scale 0 in several SQL dialects. As such the custom DECIMAL_STR with an arbitrary precision defined in the respective DBTypeFactory of a dialect is used for each cast.
- Cast to **xsd:dateTime**. The list of possible timestamp patterns is too large, and many dialects lack an effective way to check for these patterns without getting extremely verbose. Hence, it is not possible to often detect whether a value is a valid timestamp before the cast is performed. With the exception of SQL Server and Snowflake which have TRY_CAST functions, applying this function to other dialects comes with severe limitations.
- **EXISTS** is only supported when it can be translated into a

bottom-up manner, i.e. using a left join. In practice, it covers most of the cases, but sometimes a top-down evaluation is necessary.

GeoSPARQL 1.0

Starting from 4.1.0.

The following table provides a summary of the compliance of the latest version of Ontop with [OGC GeoSPARQL 1.0](#) , the standard for representing and querying geospatial linked data. The summary focuses only on the main geospatial functions and properties and unsupported features are ~~crossed out~~.

Section in OGC GeoSPARQL 1.0	Features	Co
7. Topology Vocabulary Extensions - Properties	geo:sfEquals geo:sfDisjoint geo:sfIntersects geo:sfTouches geo:sfCrosses geo:sfWithin geo:sfContains geo:sfOverlaps geo:ehEquals geo:ehDisjoint geo:ehMeet geo:ehOverlap geo:ehCovers geo:ehCoveredBy geo:ehInside geo:ehContains geo:rcc8eq geo:rcc8de geo:rcc8ec geo:rcc8po geo:rcc8tppi geo:rcc8tp geo:rcc8ntpp geo:rcc8ntppi	0
8.4. Standard Properties for Geo:Geometry	geo:dimension geo:coordinateDimension geo:spatialDimension geo:isEmpty geo:isSimple	0

	<code>geo:hasSerialization</code>	
8.5. WKT Serialization	<code>geo:wktLiteral</code> <code>geo:asWKT</code>	2/2
8.6. GML Serialization	<code>geo:gmlLiteral</code> <code>geo:asGML</code>	0
8.7. Non- Topological Query Functions	<code>geof:distance</code> <code>geof:buffer</code> <code>geof:convexHull</code> <code>geof:intersection</code> <code>geof:union</code> <code>geof:difference</code> <code>geof:symDifference</code> <code>geof:envelope</code> <code>geof:boundary</code> <code>geof:getSRID</code>	10/
9.2. Common Query Functions	<code>geof:relate</code>	1/1
9.3. Topological Simple Features Relation Family Query Functions	<code>geof:sfEquals</code> <code>geof:sfDisjoint</code> <code>geof:sfIntersects</code> <code>geof:sfTouches</code> <code>geof:sfCrosses</code> <code>geof:sfWithin</code> <code>geof:sfContains</code> <code>geof:sfOverlaps</code>	8/8
9.4. Topological Egenhofer Relation Family Query Functions	<code>geof:ehEquals</code> <code>geof:ehDisjoint</code> <code>geof:ehMeet</code> <code>geof:ehOverlap</code> <code>geof:ehCovers</code> <code>geof:ehCoveredBy</code> <code>geof:ehInside</code> <code>geof:ehContains</code>	8/8
9.5. Topological RCC8 Relation Family Query Functions	<code>geof:rcc8eq</code> <code>geof:rcc8dc</code> <code>geof:rcc8ec</code> <code>geof:rcc8po</code> <code>geof:rcc8tppi</code> <code>geof:rcc8tpp</code> <code>geof:rcc8ntpp</code> <code>geof:rcc8ntppi</code>	8/8

Several non-topological query functions use a unit of measure

URI which OGC defines under a specific namespace e.g.

`<http://www.opengis.net/def/uom/OGC/1.0/metre>`. The

latest version of Ontop currently supports the units metre, radian and degree.

R2RML

Updated for 4.2.0

The latest version of Ontop is almost fully compliant with the [R2RML](#) standard.

At the moment, it does NOT support:

- Base IRIs
- [R2RML default mapping](#) generation
- Normalization of binary SQL datatypes

For complex SQL queries (e.g. with a `GROUP BY`) in the mapping, Ontop may not be able to infer the datatype of each column if the option

`ontop.allowRetrievingBlackBoxViewMetadataFromDB` is not

enabled (disabled by default). In such a situation, it may not be able to derive the [natural RDF datatype](#) of a literal built over a column and may not apply the expected normalization. This can be partially mitigated by expliciting the RDF datatype in the mapping, but normalization would remain unapplied.

RDF 1.1

Ontop complies with [RDF 1.1](#). It types simple literals (from RDF 1.0) as `xsd:string` and literals with a language tag as

`rdf:langString`.

OWL 2 QL

RDFS

Time functions

Updated for 5.1.0.

The functions using the prefix `ofn` (`http://www.ontotext.com/sparql/functions/`) and their documentation can be found [here](#) . They accept both `xsd:date` and `xsd:dateTime` as arguments. (supported since 4.2.0).

The functions using the prefix `obdaf` (`https://w3id.org/obda/functions#`) have been introduced in 5.1.0 (see [#705](#)).

Function	Argument 1	Argument 2
<code>ofn:weeksBetween</code>	<code>xsd:date</code>	<code>xsd:date</code>
<code>ofn:weeksBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:weeksBetween</code> *	<code>xsd:date</code>	<code>xsd:dateTime</code>
<code>ofn:weeksBetween</code> *	<code>xsd:dateTime</code>	<code>xsd:date</code>
<code>ofn:daysBetween</code>	<code>xsd:date</code>	<code>xsd:date</code>
<code>ofn:daysBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:daysBetween</code> *	<code>xsd:date</code>	<code>xsd:dateTime</code>
<code>ofn:daysBetween</code> *	<code>xsd:dateTime</code>	<code>xsd:date</code>
<code>ofn:hoursBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:minutesBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:secondsBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>ofn:millisBetween</code>	<code>xsd:dateTime</code>	<code>xsd:dateTime</code>
<code>obdaf:dateTrunc</code>	<code>xsd:dateTime</code>	<code>xsd:string</code>
<code>obdaf:milliseconds-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:microseconds-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:week-from-dateTime</code>	<code>xsd:dateTime</code>	

<code>obdaf:quarter-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:decade-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:century-from-dateTime</code>	<code>xsd:dateTime</code>	
<code>obdaf:millenium-from-dateTime</code>	<code>xsd:dateTime</code>	

Combinations of argument datatypes marked with the symbol * are not supported for queries over the following data sources: Oracle and Microsoft SQL Server.

The `obdaf:[datePart]-from-dateTime` functions are supported for all dialects. They can be used to extract a specific part of the provided `dateTime` value in a numeric format (`xsd:decimal` for `milliseconds` and `microseconds`, `xsd:integer` for the remaining functions).

NOTE

The function `obdaf:week-from-dateTime` returns the ISO week index of the given date, where week 1 is considered the first week with a majority of its days in January.

The `obdaf:dateTrunc` function can be used to truncate a given `xsd:dateTime` to a new value with specified granularity. The granularity must be provided as an `xsd:string` literal. The following granularity values are supported:

- `microsecond`
- `millisecond`
- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`

- month
- quarter
- year
- decade
- century
- millennium

WARNING

Not all database systems support all granularities equally.

decade, century, and millennium are not supported by:

- AWS Athena
- Denodo (century is supported)
- MySQL (century is supported)
- MariaDB (century is supported)
- Oracle
- Presto
- SQLServer
- Snowflake
- Spark
- Trino

second is not supported by:

- Denodo

millisecond and microsecond are not supported by:

- AWS Athena
- Denodo
- MySQL
- MariaDB
- Oracle
- Presto
- Trino

PostgreSQL requires these granularities to be named milliseconds and microseconds instead.

Examples

```
obdaf:year("2023-08-16T09:00:00"^^xsd:dateTime)
"2023"^^xsd:integer
```

```
obdaf:hour("2023-08-16T09:00:00"^^xsd:dateTime)
"9"^^xsd:integer
```

```
obdaf:dateTrunc("2023-08-16T09:00:00"^^xsd:dateTime,
"month"^^xsd:string) "2023-08-
01T00:00:00"^^xsd:dateTime
```

Other functions

Updated for 5.5.0.

- Cast to `xsd:date`
- Query ID: `obdaf:queryId`

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Command Line Interface](#) [Ontop Mapping Language →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
 - Source Query
 - Target Triple Structure
 - Meta-Mapping
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Ontop Mapping Language

While Ontop is compatible with the W3C standard mapping language [R2RML](#) , it also provides its own native mapping language (OBDA).

An OBDA mapping file is a text file with the extension `.obda` and consists of two main sections:

- `PrefixDeclaration` : a list of prefix definitions used in the mapping file. Each prefix is declared by a pair of its identifier (or name) and its IRI definition.
- `MappingDeclaration` : collection of mapping assertions where each mapping assertion consists of three fields: `mappingId` , `source` and `target` . The mappingId is any string identifying the assertion, the source is an arbitrary SQL query over the database, and the target is a [triple template](#) that contains placeholders that reference column names mentioned in the source query.

The following is an example of a valid OBDA mapping file:

```
[PrefixDeclaration]
:      http://www.example.org/library#
xsd:   http://www.w3.org/2001/XMLSchema#
rdf:   http://www.w3.org/1999/02/22-rdf-syntax-ns#

[MappingDeclaration] @collection [[
mappingId      Book collection
target         :BID_{id} a :Book .
source         SELECT id FROM books

mappingId      Book title
target         :BID_{id} :title {title} .
source         SELECT id, title FROM books
]]
```

The empty lines between the two sections and between the mappings are mandatory.

Comments

To comment out a line in an OBDA mapping file, you can use the `;` character at the beginning of the line. Note that you can only comment out entire lines, not parts of them.

Source Query

The `source` query in a mapping assertion is an SQL query over the underlying relational database and as such it uses the SQL syntax of that specific database dialect. So things like quotes conventions may vary depending on the database system used. For example, in PostgreSQL double quotes are used for tables and column identifiers, while in MySQL backticks are used.

WARNING

The Ontop SQL parser only parses simple SQL queries without unions, aggregations, order by, etc. Non-parsed queries are treated as black-box views and sent directly to the database so the optimizations that Ontop can apply are limited.

Target Triple Structure

This section explains the syntax and limitations for the `target` of mapping assertions, which is an adaptation of the `Turtle` syntax. Each target triple is written like an RDF subject-predicate-object (SPO) graph.

```
target <http://www.example.org/library#BID_{id}>
rdf:type :Book .
```

[S]

```
[P]      [O]

target  <http://www.example.org/library#BID_{id}>
:title {title} .

                                [S]

[P]      [O]
```

Each triple must be separated by a space followed by a period (`s p o .`) and is composed of three nodes:

- Subject node: The subject node can be one of the following terms:
 1. IRI or blank node constant: e.g.
`<http://www.example.org/library#BID_FF125>` or `_:Library1`
 2. IRI or blank node template: e.g.
`<http://www.example.org/library#BID_{id}>` or `_{id}`
 3. IRI or blank node column: a column directly from the source query (e.g. `<{iri}>`)
- Predicate node: The predicate node can be one of the following terms:
 1. IRI constant: e.g.
`<http://www.example.org/library#title>`
 2. IRI template: e.g. `<http://www.example.org/library#{predicate}>`
 3. IRI column: a column from the source query (e.g. `<{predicate_iri}>`)

Note that the special predicate `a` is a shortcut that stands for `rdf:type` (more precisely, for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`).

- Object node: The object node can be one of the following terms:
 1. IRI or blank node constant: e.g.,
`<http://www.example.org/library#Book>`
 2. IRI or blank node template: e.g.,

- `<http://www.example.org/Author-{pid}>`
- 3. IRI or blank node column: e.g. `<{object_iri}>`
- 4. Literal constant: either an implicitly typed literal (e.g., `123` or `true` or `"John"`), an explicitly typed literal (e.g., `"John"^^xsd:string`, `"123"^^xsd:integer`) or a literal with a language tag (e.g., `"Il Trono di Spade"@it`).
- 5. Literal column: a column from the source query (e.g., `{title}`). It can also be explicitly typed (e.g., `{title}^^xsd:string`) or have a language tag (e.g., `{title}@en`).
- 6. Literal template: just like literal constants, literal templates can also be explicitly typed or have a language tag. Literal templates can be useful to create complex, arbitrary literals by concatenation (e.g., `"POINT ({longitude} {latitude})"^^geo:wktLiteral`).

IRI and blank node templates apply IRI-safe encoding to their columns, following the [R2RML standard](#). For example, if we have the IRI template

`<http://www.example.org/library#BID_{name}>` and suppose that for the `name` column we have the value `"John Library"`, then the generated IRI will be

`<http://www.example.org/library#BID_John%20Library>`.

Instead, IRI columns are not transformed, so their values are expected to already be valid IRIs.

WARNING

Literal constants, templates, and columns can either be explicitly typed or have a language tag, but the two cannot be combined. For example, the following mapping is *invalid*:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books
WHERE lang='ITALIAN'
target         :BID_{id} :title
               {title}^^xsd:string@it .
```

IRI or Blank Node Template

IRI or blank node templates are used in the target of mapping assertions for the identification of generated objects. An IRI/blank node template is a string with placeholders (e.g.

`<http://www.example.org/library#BID_{id}>`). More than one placeholder can appear in a template, which allows constructing complex paths. For example, as an IRI template:

```
mappingId      Spare parts
source         SELECT product, part, vendor FROM
product
target         <http://example.org/{vendor}/{product}/{part}> a
               :Part .
```

or as a blank node template:

```
mappingId      Spare parts
source         SELECT product, part, vendor FROM
product
target         _:{product}/{part} a :Part .
```

Prefixes in IRI or Blank Node Templates

Prefixes can be used when writing IRI or blank node templates and are replaced by their definition when Ontop parses the mappings.

Example. Assume that the following prefixes are defined:

```
: http://www.example.org/ontology1#
p: http://www.example.org/ontology2#
```

Then this mapping assertion:

```
mappingId      Example
source         SELECT col1, col2 FROM table
target         <http://www.example.org/ontology1#
```

```
{col1}> :property
<http://www.example.org/ontology2#{col2}>
```

is equivalent to this mapping assertion:

```
mappingId      Example
source         SELECT col1, col2 FROM table
target         :{col1} :title p:{col2}
```

Literal

Literal Typing

It is possible to explicitly declare the type of a literal by suffixing it with `^^` followed by the IRI of the datatype. For example:

```
mappingId      Book titles
source         SELECT id, title, edition, comment
FROM books
target         :BID_{id} :title
               {title}^^xsd:string; :edition
               {edition}^^xsd:integer; :description {comment} .
```

The type used in the mapping has to agree with the type in the ontology (if specified). If the type is not specified (for example, for the `description` property in the previous mapping), the system will look at the SQL type of the SQL column used in the mapping and will use the [Natural Mapping of SQL values](#) as defined by [R2RML standard](#) .

Language Tags

The language for a literal can be specified directly using the `@` symbol followed by the language tag. For example:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
               lang= 'ITALIAN'
target         :BID_{id} :title {title}@it .
```

WARNING

Language tags can only be constants, it is not possible to obtain them dynamically from the database. So for example, the following mapping is *invalid*:

```
mappingId      Book titles in Italian
source         SELECT id, title, lang FROM
books
target         :BID_{id} :title
               {title}@{lang} .
```

Named Graphs

By default, triples generated by a triple pattern are added to the default graph. However, it is also possible to specify a named graph by using the keyword `GRAPH` followed by an IRI constant or template and then the triple pattern in curly braces. For example:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
lang='ITALIAN'
target         GRAPH
<http://www.example.org/graphs/italian> {
:BID_{id} :title {title}@it . }
```

or, using a template for the graph name:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
lang='ITALIAN'
target         GRAPH
<http://www.example.org/graphs/{lang}> { :BID_{id}
:title {title}@it . }
```

Compact Form

Following the [Turtle](#) syntax, Ontop's native mapping format allows writing an RDF graph in a compact textual form. A set of triples sharing the same subject can be written as a predicate list, where predicate-object pairs are separated by semicolons. Similarly, a set of triples sharing the same subject and predicate can be written as an object list, where objects are separated by commas.

Predicate List: These two examples are equivalent ways of writing the triple template for an Author.

```
:Author-{ID} a :Author .
:Author-{ID} :firstName {FNAME} .
:Author-{ID} :lastName {LNAME} .
:Author-{ID} :writes :Book-{ID} .
```

```
:Author-{ID} a :Author; :firstName {FNAME};
:lastName {LNAME}; :writes :Book-{ID} .
```

Object List: These two examples are equivalent ways of writing the triple template for the *A Game of Thrones* book.

```
:A_Game_of_Thrones :title "A Game of Thrones"@en-US .
:A_Game_of_Thrones :title "Il Trono di Spade"@it .
```

```
:A_Game_of_Thrones :title "A Game of Thrones"@en-US, "Il Trono di Spade"@it .
```

Meta-Mapping

Meta-mapping assertions are syntactically the same as normal assertions, but they allow users to include variables in the targets without restrictions. This means that class and property names can be constructed dynamically from the database.

Example: Consider the following mapping assertions:

```
mappingId      mapping1
target         <{iri}> a :{value}_{code} .
source         SELECT value, iri, code FROM table1
WHERE code > 0
```

```
mappingId      mapping2
target         <{iri}> :{role}_{code} {value} .
source         SELECT value, iri, code, role FROM
table1 WHERE code > 0
```

Suppose we also have a table named `table1` that the mapping assertions refer to:

iri	value	code	role
iri1	A	1	P
iri2	B	2	P
iri3	A	2	Q
iri4	B	2	Q

Then `mapping1` will generate the following triples:

```
iri1 a :A_1 .
iri2 a :B_2 .
iri3 a :A_2 .
iri4 a :B_2 .
```

And `mapping2` will generate the following triples:

```
iri1 :P_1 A .
iri2 :P_2 B .
iri3 :Q_2 A .
iri4 :Q_2 B .
```


Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
 - HTTP API
 - Example
 - Entries
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Predefined query endpoint (beta)

Since 4.1.0.

The predefined query endpoint enables the specification of custom low-latency Web APIs in a declarative manner. It relies on two main components:

1. A JSON-based configuration file, describing the API interface, its parameters and optionally a JSON-LD frame for shaping the returned data.
2. A SPARQL query in charge of retrieving and perhaps transforming the data. The values of the API parameters are plugged as constants into the SPARQL query.

WARNING

At the moment only CONSTRUCT queries are supported.

This endpoint brings one important benefit in terms of performance with respect to the SPARQL endpoint: it is able to reuse reformulations from similar queries. This is particularly valuable for "lookup" requests involving many optional triples, and for which latency is critical.

This endpoint can be deployed using the command `ontop endpoint` or the Docker image `ontop/ontop`.

HTTP API

```
GET /predefined/{queryId}?{param1}={value1}&{param2}={value2}&{param3}={value3}
```


Example

Configuration

```

{
  "queries": {
    "lodging": {
      "queryType": "graph",
      "name": "Lodging businesses",
      "description": "JSON-LD snippets for
lodging businesses, based on schema.org.",
      "return404IfEmpty": true,
      "resultStreaming": false,
      "parameters": {
        "id": {
          "description": "Internal
identifier of the lodging business",
          "type": "xsd:string",
          "safeForRandomGeneration":
true,
          "required": true
        },
        "language": {
          "description": "Language (e.g.
it, de, en, ru). Default: en",
          "type": "xsd:string",
          "safeForRandomGeneration":
false,
          "required": false
        }
      },
      "frame": {
        "@context": "https://schema.org",
        "@type": [
          "LodgingBusiness"
        ]
      }
    }
  }
}

```

Predefined queries

```
[lodging]
query=""
BASE <http://noi.example.org/data/accommodation/>
PREFIX schema: <http://schema.org/>
PREFIX : <http://noi.example.org/ontology/odh#>
CONSTRUCT {
    ?h a schema:LodgingBusiness ; schema:name ?
nameStr ; schema:priceRange ?priceRange .
}
WHERE {
    # Convert incoming parameters
    BIND (IRI(?id) AS ?h)
    BIND (COALESCE(?language, "en") AS ?l)

    ?h a schema:LodgingBusiness .

    OPTIONAL {
        ?h schema:name ?name .
        BIND(str(?name) AS ?nameStr)
        FILTER(lang(?name) = ?l)
    }
    OPTIONAL {
        ?h schema:priceRange ?priceRange .
    }
}
""
```

Cache for the JSON-LD contexts

Optionally, JSON-LD context documents can be stored in a cache and adapted if needed. When they are not cached, these documents are directly fetched from the Web.

```
{
  "https://schema.org": {
    "@context": {
      "@vocab": "http://schema.org/"
    }
  }
}
```

json

Request

```
curl http://localhost:8080/predefined/lodging?
id=32E7BE648E7B11D181AB006097B896BA&language=de
```

Results:

```
{
  "@context": "https://schema.org",
  "@id":
"http://noi.example.org/data/accommodation/32E7BE648E7

  "@type": "LodgingBusiness",
  "name": "Alpenblick - Gasthaus"
}
```

Entries

Top-level key

Key	Type	Description
<code>queries</code>	Map of predefined queries	Groups predefined queries by id.

Query keys

Key	Type	Description
<code>queryType</code>	String	At the moment, only <code>graph</code> is supported. Corresponds to CONSTRUCT queries.
<code>name</code>	String	Name of the predefined query.
<code>description</code>	String	Description of the predefined query.
		If true, returns a

<code>return404IfEmpty</code>	Boolean	404 status code when the result set is empty.
<code>resultStreaming</code>	Boolean	If true, uses the chunked transfer encoding.
<code>parameters</code>	Map of parameters	Groups parameters by id.
<code>frame</code>	JSON object	JSON-LD frame) for shaping the result set when the JSON-LD format is selected.

Parameter keys

Key	Type	Description
<code>description</code>	String	Description of the parameter.
<code>type</code>	String	Type of the parameter. Can be <code>iri</code> for an IRI, any XSD datatype like <code>xsd:string</code> with the prefix <code>xsd:</code> or the IRI string of an RDF datatype.
		If true, replacing the parameter value by a random value for generating a "reference" reformulated query can be considered as safe. Replacing afterwards

<code>safeForRandomGeneration</code>	Boolean	the random value but the real parameter value in the reference reformulated query should produce the expected result. Usually, identifiers are safe while languages are not because the columns to consider differ from a language to another.
<code>required</code>	Boolean	If true, a value must be given for this parameter in the HTTP request.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Ontop Mapping Language](#)

[Query logging →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)

Query logging

- Example
- Entries
- Default settings
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Query logging

Since 4.1.0.

Query logging, disabled by default, prints one-line JSON objects into the standard output.

Its JSON format follows some Elasticsearch conventions and works nicely with the corresponding stack (FileBeat, ES, Kibana).

Example

json

```
{
  "@timestamp": "2023-10-13T14:02:05.973Z",
  "message": "query:all",
  "application": "ontop-odh",
  "payload": {
    "queryId": "36ade6c6-f091-4af2-9828-ee6316e191f4",
    "classesUsedInQuery": [
      "http://schema.org/Hotel"
    ],
    "propertiesUsedInQuery": [
      "http://schema.org/name"
    ],
    "tables": [
      "\"v_accommodationsopen\""
    ],
    "reformulationDuration": 50,
    "reformulationCacheHit": false,
    "user": "joe",
    "groups": "admins,users",
    "roles": "my_app_write,my_app_reader",
    "httpHeaders": {
      "referer": "http://localhost:8080/"
    },
    "extractedQueryTemplate": {
      "hash":
"de167012e1ce3cf9dce66a986a3343d883a1f319f800132565b61"
```

```
      "parameters": {
        "v0": "\"Residence Tuberis\""
      }
    },
    "sparqlQuery": "PREFIX schema:
<http://schema.org/>\nSELECT * WHERE {\n  ?h a
schema:Hotel ; schema:name ?name .\n  FILTER
(langMatches(lang(?name), 'en'))\n  FILTER(str(?
name) = \"Residence Tuberis\")\n} \nLIMIT 10",
    "reformulatedQuery": "ans1(h,name)\nCONSTRUCT
[h, name] [name/\"Residence Tuberis\"^^@en,
h/RDF(http://noi.example.org/data/accommodation/{
(VARCHARToTEXT(Id1m148)),IRI)]\n  NATIVE
[Id1m148]\nSELECT v1.\"Id\" AS \"Id1m148\"\nFROM
\"v_accommodationsopen\" v1\nWHERE ('Residence
Tuberis' = v1.\"AccoDetail-en-Name\" AND
'HotelPension' = v1.\"AccoTypeId\")\nLIMIT 10\n",
    "executionBeforeUnblockingDuration": 8,
    "executionAndFetchingDuration": 30,
    "totalDuration": 80,
    "resultCount": 10
  }
}
```

Entries

Key	Type	
@timestamp	Timestamp	
message	String	Type of the message. Its value is the query. If an exception occurs, its value is the exception message.
application	String	Controlled by the application. Its value is the application name.
payload	JSON object	Gathers the execution details.
queryId	UUID	Unique to each query.
classesUsedInQuery	Array of IRIs	IRIs of the classes used in the query.

<code>propertiesUsedInQuery</code>	Array of IRIs	IRIs of the pre query.
<code>tables</code>	Array of Strings	Names of the
<code>reformulationDuration</code>	Integer	Query reform
<code>reformulationCacheHit</code>	Boolean	True if the ref
<code>user</code>	String	Since 5.2.0. U
<code>groups</code>	String	Since 5.2.0. U Disabled by c
<code>roles</code>	String	Since 5.2.0. U Disabled by c
<code>httpHeaders</code>	JSON object	Values of a s this list is emp the property <code>ontop.query</code> to true.
<code>extractedQueryTemplate. hash</code>	String	Hash of the q removing ext similar querie
<code>extractedQueryTemplate. parameters</code>	JSON object	Constants ext
<code>sparqlQuery</code>	String	SPARQL quer
<code>reformulatedQuery</code>	String	Includes the S
<code>executionBefore UnblockingDuration</code>	Integer	Duration betw query is sent
<code>executionAndFetching Duration</code>	Integer	Duration betw query is sent
<code>totalDuration</code>	Integer	Total durator
<code>resultCount</code>	Integer	Number of re

Default settings

```
# Query logging is disabled by default
ontop.queryLogging=false
```



```
# Application name (needed for logging)
ontop.applicationName=Ontop

# Includes the SPARQL query string into the query
log
ontop.queryLogging.includeSparqlQuery=true
# Includes the reformulated query into the query
log
ontop.queryLogging.includeReformulatedQuery=false
# Includes classes and properties into the query
log
ontop.queryLogging.includeClassesAndProperties=true

# Includes DB tables/views into the query log
ontop.queryLogging.includeTables=true
# Includes the user ID, his/her groups and roles
ontop.queryLogging.includeUserInfo=false
# Provides separated message at different phases
(after reformulation, result set unblocked, last
result fetched)
ontop.queryLogging.decomposition=false
# Sets that merged messages are only inserted when
decomposition is disabled
ontop.queryLogging.decompositionAndMergingMutuallyExcl
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Predefined query endpoint \(beta\)](#)

[HTTP caching →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Remarks
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

HTTP caching

Since 4.1.0.

HTTP caching allows to reduce the load on the SPARQL endpoint and to provide faster responses to clients. It is particularly useful when many clients issue the same queries and data is not updated at a high frequency.

To enable HTTP caching, add the property `ontop.http.cacheControl` in the properties file. This will cause the insertion of a `Cache-Control` HTTP header in successful responses, with the specified value.

Example

```
ontop.http.cacheControl=max-age=60, stale-while-revalidate=20, stale-if-error=300
```

This entry informs the cache that values can be considered as fresh for 60 seconds. After this period, the cache remains allowed to serve cached results for 20 more seconds while revalidating the requests in the background, or for 300 more seconds in case of error with the upstream server.

Ontop does not embed a HTTP cache. We recommend setting up a HTTP cache like [Varnish](#) or [Nginx](#) above the Ontop endpoint.

Remarks

- Most caches (e.g. Nginx but also Web browsers) do not cache POST requests by default. Since most SPARQL queries are sent

with the POST method, it is important to turn this option on.
This option is safe as Ontop is a read-only system.

- Consider using the GET method for sending SPARQL queries so as to take advantage from Web browser caches.
- For an example of configuration for Nginx, see [this file](#) .

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Query logging](#)

[Configuration keys →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Configuration keys

Here is a list of the configuration keys which can be set in the properties file.

Key	Type	Description

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← [HTTP caching](#) [Lenses](#) →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys

Lenses

- Example
- Document root
- Lens
- Constraints

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Lenses

Since 4.2.0 (was experimental in 4.1.x)

Prior to 5.0.0, lenses were named Ontop views.

Lenses are relational views defined at the level of Ontop and unknown to the underlying database. Lenses can be defined from database relations and from other lenses.

As database relations, lenses have a name which can be used in the source part of the mapping entries. They are specified in a separate file that can be provided to Ontop through a dedicated parameter (`--lenses` for the [CLI commands](#) that support it, `ONTOP_LENSSES_FILE` for the [Docker image](#)).

At the moment, 5 types of lenses are available:

1. [Basic lenses](#) (defined over one base relation)
2. [Join lenses](#) (defined over multiple base relations)
3. [SQL lenses](#) (defined from an arbitrary SQL query)
4. [Union lenses](#) (defined as a union of multiple base relations)
5. [Flatten lenses](#) (defined as an unnest operation over one base relation)

Don't use lenses in complex source SQL queries

The Ontop mapping SQL parser only parses simple forms of SQL queries (without unions, aggregations, limits, order by, etc.). Non-parsed queries are treated as black-box views, that is as strings that are injected into the final SQL queries sent to the database. If some lenses appear in these black-box views, the resulting SQL queries will be rejected by the database because they refer to relations it does not know.

One interesting feature of lenses is that you can specify additional

[constraints](#) holding on them (in addition to the ones that can be inferred from base relations). The constraints can be:

- [Unique constraints](#)
- [Other functional dependencies](#)
- [Foreign keys](#)
- [Non-null constraints](#) (columns that do not include null values)
- [IRI-safe constraints](#) (columns on which the R2RML safe encoding has no effect). *Since 5.0.2.*

Tutorial

To see lenses used in practice, check [the tutorial](#)

Example

```

{
  "relations": [
    {
      "name":
["\"lenses\"", "\"hr\"", "\"persons\""],
      "baseRelation":
["\"hr\"", "\"persons\""],
      "filterExpression": "\"firstName\" IS
NOT NULL AND \"lastName\" IS NOT NULL",
      "columns": {
        "added": [
          {
            "name": "\"fullName\"",
            "expression":
"CONCAT(UPPER(\"firstName\"), ' ', \"lastName\")"
          }
        ],
        "hidden": [
          "\"firstName\"",
          "\"lastName\""
        ]
      },
      "uniqueConstraints": {
        "added": [
          {
            "name": "uc2",

```

```

        "determinants":
["\"ssn\""]
    }
]
},
"otherFunctionalDependencies": {
    "added": [
        {
            "determinants":
["\"regionOfResidence\""],
            "dependents":
["\"countryOfResidence\""]
        }
    ],
    "foreignKeys": {
        "added": [
            {
                "name": "fk1",
                "from":
["\"regionOfResidence\""],
                "to": {
                    "relation":
["\"geo\"", "\"regions\""],
                    "columns":
["\"reg_id\""]
                }
            }
        ],
        "nonNullConstraints": {
            "added": [
                "\"email\""
            ],
        },
        "iriSafeConstraints": {
            "added": [
                "\"ssn\""
            ],
        },
        "type": "BasicLens"
    },
    {
        "name": [
            "\"lenses\"",
            "\"rooms\""
        ],
    },

```

```
      "join": {
        "relations": [
          ["\"rooms\""],
          ["\"lenses\"", "\"hotels\""]
        ],
        "columnPrefixes": [
          "r_",
          "h_"
        ]
      },
      "filterExpression":
        "\"r_hotel_id\"=\"h_id\" AND (\"h_stars\" = '***'
        OR \"h_price\" = '€€€') AND \"r_guests\" = 2",
      "columns": {
        "added": [
        ],
        "hidden": [
        ]
      },
      "type": "JoinLens"
    },
    {
      "name":
        ["\"lenses\"", "\"geo\"", "\"top_region\""],
      "query": "SELECT \"regionOfResidence\"
        AS \"region\", COUNT(*) FROM \"hr\".\"persons\"
        GROUP BY \"regionOfResidence\" ORDER BY COUNT(*)
        DESC LIMIT 1",
      "type": "SQLLens"
    }
  ]
}
```

Document root

The lenses document has the following JSON structure:

```
{
  "relations": [Lens]
}
```

Key	Type

relations

Array of

Lens

-s

Lens

Common fields

All the lenses accept the following fields (most of them are optional):

```
{
  "name": [String],
  "uniqueConstraints": {
    "added": [UniqueConstraint]
  },
  "otherFunctionalDependencies": {
    "added": [OtherFunctionalDependency]
  },
  "foreignKeys": {
    "added": [ForeignKey]
  },
  "nonNullConstraints": {
    "added": [String]
  },
  "iriSafeConstraints": {
    "added": [String]
  },
  "type": String
}
```

json

Key	Type
name	Array of Strings
uniqueConstraints	JSON Object
uniqueConstraints.added	Array of UniqueConstr
otherFunctionalDependencies	JSON Object
otherFunctionalDependencies.added	Array of OtherFunctionalDeper

	s
foreignKeys	JSON Object
foreignKeys.added	Array of ForeignKey
nonNullConstraints	JSON Object
nonNullConstraints.added	Array of Strings
iriSafeConstraints	JSON Object
iriSafeConstraints.added	Array of Strings
type	String

BasicLens

A basic lens is defined from one base (parent) relation, over which it can apply a filter, an extended projection and additional constraints.

In addition to the common fields, basic lenses accept the following ones:

```
{
  "baseRelation": [String],
  "columns": {
    "added": [AddedColumn],
    "hidden": [String]
  },
  "filterExpression": String,
```

```
"type": "BasicLens"
}
```

Key	Type	Description
<code>baseRelation</code>	Array of Strings	Name components of the base relation (with correct quoting)
<code>columns</code>	JSON Object	Optional (since 5.0.2)
<code>columns.added</code>	Array of <code>AddedColumns</code>	
<code>columns.hidden</code>	Array of Strings	Names of the columns from the base relation to be projected away (with correct quoting)
<code>filterExpression</code>	String	Expression expressed in the SQL dialect of the data source. Can only refer to columns from the base relation, not to added columns. Can be empty. Optional

`AddedColumn`

Added columns have the following definition:

```
{
  "name": String,
}
```

json

```

    "expression": String
  }

```

Key	Type	Description
name	String	New column name (with correct quoting)
expression	String	SQL expression defining the column. Can only refer to columns from the base relations, not to added columns

JoinLens

A join lens is defined from multiple base relations, over which it can apply a filter (joining condition), an extended projection and additional constraints.

A prefix is assigned to each base relation and is added as a prefix to their column names. This allows to avoid conflicts due to columns with the same names in base relations.

In addition to the common fields, join lenses accept the following ones:

```

{
  "join": {
    "relations": [[String]],
    "columnPrefixes": [String]
  },
  "columns": {
    "added": [AddedColumn],
    "hidden": [String]
  },
  "filterExpression": String,
  "type": "JoinLens"
}

```

Key	Type	Descripti
join	JSON Object	

<code>join.relations</code>	Array of arrays of Strings	Arrays of the components each base re (with correct quoting)
<code>join.columnPrefixes</code>	Array of Strings	Prefix for each base relation applied on its column name. Follows the same order as <code>join.relations</code>
<code>columns</code>	JSON Object	Optional (since 5.0.2)
<code>columns.added</code>	Array of <code>AddedColumns</code>	
<code>columns.hidden</code>	Array of Strings	Names of the columns from base relation to be projected (with correct quoting)
<code>filterExpression</code>	String	Expression expressed in SQL dialect of data source. It can only refer to prefixed columns from the base relations, not added columns. Can be empty. Optional

SQLLens

A SQL lens is defined from an arbitrary SQL query. While expressive, it also comes with important restrictions. When applicable, other types of lenses should be used instead.

Avoid referring to lenses in the SQL query

As Ontop uses the same parser as from the mapping

source queries, the same restriction apply: non-parsed queries will be treated internally as black-box views and will fail. Please consider using other types of lenses if possible.

No unique constraint and foreign key inferred from the base relations

Please consider using other types of lenses if possible.

In addition to the [common fields](#), SQL lenses accept the following ones:

```
{
  "query": String,
  "type": "SQLLens"
}
```

json

Key	Type	Description
query	String	SQL query

UnionLens

Since 5.1.0

A union lens is defined from multiple base relations that share attributes with exactly the same names and types. The relations will be merged with each other, concatenating their contents.

When defining a union lens, a *"provenance column"* can be determined to hold, for each data entry, the name of the base relation it originates from.

In addition to the [common fields](#), union lenses accept the following ones:

```
{
  "unionRelations": [[String]],
}
```

json

```
"makeDistinct": boolean,  
"provenanceColumn": String,  
"type": "UnionLens"  
}
```

Key	Type	Description
unionRelations	Array of arrays of Strings	Arrays of the name components of each base relation (with correct quoting).
makeDistinct	boolean	Determines, if the final resulting union should be made distinct.
provenanceColumn	String	The name of the column that should contain the base relation each entry originates from. If not provided, provenance information will not be included in the result.

FlattenLens

Since 5.1.0

A flatten lens is defined from one base (parent) relation that contains an array-like data structure in one of its fields. The array is flattened into multiple rows, where each row contains a single item from the flattened array in the `new` column. Columns of the base relation not included in the `kept` list will be discarded when flattening the array.

In addition, a `position` column can be included in the lens, providing a unique index for each flattened row in its parent relation.

NOTE

The flatten operation is only performed on the "outer-most" array layer. Multi-dimensional arrays will have their dimensionality reduced by 1.

In addition to the [common fields](#), flatten lenses accept the following ones:

json

```
{
  "flattenedColumn": {
    "name": String,
    "datatype": String
  },
  "columns": {
    "kept": [String],
    "new": String,
    "position": String
  },
  "type": "FlattenLens"
}
```

Key	Type	Description
<code>flattenedColumn</code>	JSON Object	Identifies the column that is to be flattened.
<code>flattenedColumn.name</code>	String	The name of the column that is to be flattened.
<code>flattenedColumn.datatype</code>	String	The type of the column that is to be flattened.
<code>columns</code>	JSON Object	Defines the columns of the output relation.
<code>columns.kept</code>	Array of Strings	The names of the columns from the base relation that should be included in the output.
		The name of the newly

<code>columns.new</code>	String	created column that should hold the elements of the flattened array.
<code>columns.position</code>	String	The name of the newly created column that should hold the index of each flattened element in its source list. If not provided, no position column will be included.

Due to various limitations in the language definitions, the FlattenLens is currently not equally supported for all dialects. The table below lists, in detail, the level of support for each dialect. "Flatten" defines if the flatten lens is supported by the dialect, "position" defines if the `position` column can be provided, and "Infer base type" indicates if Ontop is able to infer the type of the flattened output column if the input is an array type. "Array Type" and "JSON Type" indicate if the flatten lens is supported over array-like types (`ARRAY`, `ARRAY<T>`, `LIST`, `T[]` etc.) and JSON-arrays (either as `JSON` type or as `VARCHAR`) respectively.

Dialect	Flatten	Position	Infer base type	Array Type
AWS Athena	✓	✓	✓	✓
AWS DynamoDB	✗	✗	✗	✗
AWS Redshift	✓	✓	✗	✓
BigQuery	✓	✓	✓	✓

DB2	✗	✗	✗	✓
Databricks	✓	✓	✓	✓
Denodo	✗	✗	✗	✓
Dremio	✓	✗	✗	✓
DuckDB	✓	✓	✓	✓
MariaDB	✓	✓	✗	✗
MS SQLServer	✓	✗	✗	✗
MySQL	✓	✓	✗	✗
Oracle	✓	✓	✗	✗
PostgreSQL	✓	✓	✓	✓
Presto	✓	✓	✓	✓
Snowflake	✓	✓	✗	✓
SparkSQL	✓	✓	✓	✓
TDengine	✗	✗	✗	✗
Trino	✓	✓	✓	✓

Constraints

UniqueConstraint

json

```
{
  "name": String,
  "determinants": [String]
}
```

Key	Type	Description
<code>name</code>	String	Name of the unique

		constraint
determinants	Array of Strings	Column names (with correct quoting)

OtherFunctionalDependency

Useful for dealing with denormalized data, where unique constraints cannot be applied.

```
{
  "determinants": [String],
  "dependents": [String]
}
```

json

Key	Type	Description
determinants	Array of Strings	Column names (with correct quoting) that determine the values of dependent columns
dependents	Array of Strings	Column names (with correct quoting) whose values are determined by determinant columns

ForeignKey

```
{
  "name": String,
  "from": [String],
  "to": {
    "relation": [String],
    "columns": [String]
  }
}
```

json

Key	Type	Description
name	String	Name of the foreign key
	Array	Source columns (with correct

<code>from</code>	of Strings	quoting)
<code>to</code>	JSON Object	
<code>to.relation</code>	Array of Strings	Name components of the target relation (with correct quoting)
<code>to.columns</code>	Array of Strings	Target columns (with correct quoting). Same order as for the source columns

NotNullConstraint

```
{
  "added": [String]
}
```

json

Key	Type	Description
<code>added</code>	Array of Strings	List of names of non-nullable columns. One String per column

IRISafeConstraint

```
{
  "added": [String]
}
```

json

Key	Type	Description
<code>added</code>	Array of Strings	List of names of IRI-safe columns. One String per column

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

Introduction

Key concepts

Getting started

Command Line Interface

Standards compliance

Advanced

Ontop Mapping Language

Predefined query endpoint (beta)

Query logging

HTTP caching

Configuration keys

Lenses

Data sources ►

Troubleshooting

FAQ

General

Protégé

Meta

Glossary

Release notes

FAQ

Some Frequently Asked Questions

General

Which mapping syntaxes are supported?

We support [R2RML](#) and [our own mapping syntax](#), which can be automatically converted into R2RML using Ontop CLI command

```
ontop mapping to-r2rml
```

Is it open source?

Yes! Ontop is available under the Apache License, Version 2.0.

How can I report a bug or ask other questions?

You can get in touch with us through [our support channels](#) and [contribute to the project](#) by, for instance, filing a [bug report](#).

Does Ontop support SPARQL Update?

No, as for now, Ontop is a read-only system, it does not support [SPARQL Update queries](#) but only SELECT, CONSTRUCT, ASK and DESCRIBE queries. For more details, see [the compliance page](#).

How to set the log level to debug?

For both the Docker image and the CLI, set the environment variable `ONTOP_LOG_LEVEL` to `debug`.

Protégé

No suitable driver found for .. (driver)

You need to download and install the right JDBC driver in Protégé. This is required in order to query using SPARQL and use other features of Ontop. To add a new JDBC Driver:

- in Protégé, go to *Preference -> JDBC Drivers* tab;
- add a new entry;
- type a description, e.g., *PostgreSQL JDBC Driver*;
- type the driver's class string, e.g., *org.postgresql.Driver*;
- type or select the location of the driver's JAR file, e.g., *postgresql-8.4-701.jdbc4.jar*;
- click OK .

See also [Setting up JDBC Drivers](#) and [JDBC Information](#) .

No "JDBC drivers" tab in the preferences

You probably can see the following message in the Protégé console:

```
org.osgi.framework.BundleException: Activator
start error in bundle org.protege.osgi.jdbc.prefs
[20].)
org.protege.osgi.jdbc.RegistryException:
java.lang.ClassNotFoundException: org.h2.Driver
```

Please go to the *Preferences* and click on "*Reset preferences*".

Protégé does not show the Ontop tabs

To see the tabs go to *Windows -> tabs -> Ontop Mappings and Ontop SPARQL*.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- VKG
- Mapping
- OBDA
- Ontology
- RDF
- RDFS
- SPARQL
- OWL
- R2RML
- Release notes

Glossary

VKG

Virtual Knowledge Graph

Mapping

OBDA

Ontology-Based Data Access

Ontology

RDF

RDFS

SPARQL

SPARQL Query

SPARQL Endpoint

OWL

R2RML

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary

Release notes

- 5.5.0 (February 14, 2026)
- 5.4.0 (September 29, 2025)
- 5.3.1 (September 29, 2025)
- 5.3.0 (February 18, 2025)
- 5.2.1 (November 24, 2024)
- 5.2.0 (August 16, 2024)
- 5.1.2 (January 17, 2024)
- 5.1.1 (November 22, 2023)
- 5.1.0 (August 15, 2023)
- 5.0.2 (March 9, 2023)
- 5.0.1 (January 29, 2023)

Release notes

5.5.0 (February 14, 2026)

New features

- Support paths to files/directories as relation names in FROM clauses in mappings [#905](#)
- Common Table Expression in SQL serialization [#911](#)
- Basic operations on constant time intervals [#916](#)
- New SPARQL function: `obdaf:queryId` [#924](#)

Refactoring

- Refactoring IQTree creation and transformation [#906](#)
- SQLQueryParser simplifications [#907](#)
- DB metadata extraction adapted for DuckDB 1.4 [#915](#)

[Milestone on Github](#)

5.4.0 (September 29, 2025)

New features

- New query unfoldr, performing query unfolding in two phases for handling abstract SPARQL queries more efficiently, added [#846](#)
- Support for `rdfs:subClassOf*` property path added [#832](#)
- Bottom-up implementation of `EXISTS`, supporting most cases for `EXISTS`, added [#853](#) and [#861](#)
- Support for TDEngine added [#875](#) and [#877](#)
- Support for multi-line target queries in OBDA files added [#874](#)

New optimization

- Case-insensitive substring matching simplified [#882](#)

Refactoring

- Robust to primary keys with NULLable attributes added #859
- `IQTree` and `QueryNode` code cleaned up #862
- Numeric binary operators refactored #893
- More Trino functions recognized #849

Bug fixes

- `ORDER BY` lifted above `DISTINCT` when necessary #847
- Foreign key inference in lenses preserved in the presence of IRI safeness declarations #880
- Correct datatypes declared for all geospatial related functions

[Milestone on Github](#)

5.3.1 (September 29, 2025)

New features

- Database init script added #856
- `ST_Distance` function for Trino added #845
- H2 information schema tables are now classified as system tables #869

New optimization

- Conjunctive query containment check now handles NULLable columns #872

Refactoring

- `CHARACTER VARYING` and `VARCHAR` are now treated the same in Dremio #881

Bugfixes

- Arity violation fixed #866
- DuckDB catalogs are now taken into account #867
- Non-renamed variable in `NullableFDSelfLJOptimizer` fixed

#900

- Missing padding with nulls when simplified left joins fixed
- Long artificial `LIMIT` value reduced for supporting Dremio 26

[Milestone on Github](#)

5.3.0 (February 18, 2025)

Major dependency update

- RDF4J updated from 4.x to 5.1.0 #819
- OWLAPI updated to 5.5.1 #819

New optimizations

- Unique constraint inference for unions improved in presence of disjoint IRI templates
- Inference of strict dependents as a lightweight alternative to functional dependencies added

Bugfix

- `STRSTARTS` handling of IRI template strings fixed

5.2.1 (November 24, 2024)

- Column info extraction for Oracle made robust to latency #827
- Implement RDF4J `Query.getDataset()` and `Query.setDataset()`
- Exponentially slow removal of variables of a large BGP after applying a distinct fixed

Apache Commons IO dependency updated

5.2.0 (August 16, 2024)

New features

- Extract user information (ID, group, role) from HTTP headers and make it available to special SQL functions #753

- Insert user information into the query log [#768](#)
- Compression option added to the materialization [#797](#)

Optimizations

- Left join optimizations:
 - Pruning the right child when unused and cardinality doesn't matter [#752](#)
 - More non-nullability constraints extracted from the ancestors for transferring self-left-joins based on functional dependencies [#761](#)
 - Self-left-join elimination with nullable determinants of functional dependencies [#783](#)
 - Self-left join elimination on nullable functional dependency determinants made robust to provenance variables and trivial functional dependencies [#804](#)
- Unique constraint and variable non requirement inference with distinct nodes improved [#756](#)
- Simplification of conjunctions and disjunctions of equalities [#755](#)
- Propagation of foreign keys from basic lenses to parents [#759](#)
- Functional dependency inference for unions improved [#760](#)
- Optimization to the SPARQL `strStartsWith` function [#801](#)
- IRI decomposition with more than 1 argument when some arguments are nullable enabled [#809](#)
- Removal of distincts below joins when there is a `limit 1` above
- Year extraction from timestamps with DuckDB optimized

Refactoring

- T-mapping and related optimizations refactored [#769](#)
- B-node label anonymization pushed into the IQTree [#772](#)
- Handling environment variables with the CLI refactored [#814](#)
- Robust simplification for GeoSPARQL functions [#808](#)
- Wrapping a values node into a lens at mapping processing time added (disabled by default; see property `ontop.wrapMappingValuesNodesInLenses`)
- Non-nullable variables are not used anymore for right provenance from a left join (was preventing some optimizations)

Bug fixes

- `<expression> = true` issue with SQLServer and Oracle fixed [#758](#)
- Connection test for Trino and H2 fixed [#767](#)
- Mysql datetime conversion is now encoded to UTF8 to avoid collation error [#779](#)
- Missing spaces in string format pattern with MySQL added [#784](#)
- Configuration classes now use injector suppliers [#794](#)
- MS SQL Server identifiers are now treated not case-sensitive (even if quoted) [#805](#)
- Incorrect time zone format in xsd:dateTime for Redshift's `TIMESTAMP` data type fixed [#807](#)
- Database constraints were ignored by Ontop CLI [#777](#)
- Error with the CLI during the R2RML to OBDA mapping conversion fixed [#799](#)
- `TIMESTAMP WITH TIME ZONE` is now recognized for H2
- Typing nulls for `DATE` with Dremio fixed
- Missing minutes from the DuckDB time zone added
- Invalid URNs for the triple patterns generated when converting the mapping to R2RML fixed
- Error with constant in order-by involving a VALUES node fixed
- Missed duplicate elimination for some overlapping IRI templates fixed
- Incorrect content types returned for CSV and TSV results of a SPARQL query fixed

[Milestone on Github](#)

5.1.2 (January 17, 2024)

Bug fixes

- Conflict between columns in case of a self-join in the bootstrapper [#786](#)
- Handling of casts for different versions of MySQL and MariaDB [#787](#)

Several dependencies updated, including JQuery ([#780](#))

5.1.1 (November 22, 2023)

New features

- Option to expose system tables added [#744](#)
- Standard deviation and variance functions added [#735](#)
- Option to disable the registration of custom SPARQL aggregation functions [#746](#)

Bug fixes

See <https://github.com/ontop/ontop/milestone/23?closed=1>

5.1.0 (August 15, 2023)

New features

- Support for [flatten lenses](#) added (*beta*) [#622](#) , [#625](#) , [#628](#) , [#629](#) , [#633](#) , [#635](#) , [#636](#) , [#637](#) , [#638](#) , [#639](#) , [#640](#) , [#641](#) , [#642](#) , [#643](#) , [#661](#) , [#715](#)
- Support for [union lenses](#) added (*beta*) [#612](#)
- Support for XSD cast functions added [#630](#)
- Support for date-trunc and more date extraction functions added [#705](#)
- Support for a separate RDF file for facts added [#666](#)
- Options to ignore invalid mapping entries [#631](#) and lenses [#709](#) added
- Partial support for regular expressions in SQL Server added [#702](#)
- Support for DynamoDB using the CData connector added (*beta*) [#693](#)

Optimizations

- Upwards propagation of lens unique constraints added [#668](#)
- Left join optimizations:
 - Reducing left joins to inner joins when the right child has a left join [#662](#)
 - Merging left joins nested on the left [#663](#) , [#686](#)
 - Robustness to nullable joining columns [#667](#)
 - Pruning the right of a left join when not contributing

- VALUES node decomposition added [#680](#)
- Functional dependency inference added [#681](#)
- Reduction of aggregation nodes with no aggregation functional terms to distinct added
- Transitive closure of functional dependencies added [#732](#)

Refactoring

- Substitutions refactored [#616](#)
- Typing for math operators added [#706](#)
- QuotedIDFactory refactored [#644](#)
- Removal of non-required variables improved [#669](#)
- Support for the STRING datatype for DuckDB added [#646](#)
- Support for SAMPLE at the database level added [#660](#)

Bug fixes

- Support LIMIT/OFFSET in SQL Server improved [#676](#) , [#679](#) , [#682](#) , [#721](#)
- Extraction of unique constraints in Oracle made more consistent [#695](#)
- Metadata extraction for DuckDB made more robust [#687](#)
- Missing casts when using Bnode templates added [#657](#)
- Invalid SPARQL UNION translation fixed [#651](#)
- Empty results for CONSTRUCT queries with unions and different variables fixed [#652](#)

Documentation

- Dedicated pages for each supported database (for instance [AWS Athena](#))
- [Tutorial about lenses](#)
- [Tutorial about materialization](#)

5.0.2 (March 9, 2023)

New features

- Support for Trino [#599](#) , PrestoDB [#601](#) , AWS Athena [#602](#) , DuckDB [#604](#) , AWS Redshift [#605](#) and Google BigQuery [#615](#)

added

- Declare some columns as IRI-safe in lenses [#587](#)
- Support for arbitrary JDBC properties added [#606](#)

Bugfixes

- Missing TLS cypher suite JRE modules added to the Docker image
- More system schemas are now ignored in Oracle [#591](#)

Tooling

- [Scaffolding tool](#) created to make it easier to implement a new SQL dialect adapter.

5.0.1 (January 29, 2023)

New feature

- DB metadata extraction for MS Access added [#589](#)
- JRE-embedded Ontop-Protégé bundles with Protégé-5.6.0-beta-2 are now provided

Refactoring

- Support for Spark SQL improved and tests added in the CI
- All the tables (besides excluded ones) in Oracle are now extracted, not just the user's ones

Several dependencies updated.

5.0.0 (December 31, 2022)

Important changes

- Java 11 or newer is required
- Ontop's Protégé plugin now requires at least Protégé 5.6-beta
- RDF4J bindings updated to 4.1.0
- The module `ontop-owlapi` is not published anymore. We recommend using `ontop-rdf4j` instead
- RDF4J workbench modules removed

- Docker image moved to `ontop/ontop`
- Ontop views renamed [lenses](#)

New features

- Support for SPARQL rules for the extending the VKG added [#576](#)
- Support for SPARQL `FROM` and `FROM NAMED` constructs (also part of [#576](#))
- The [Docker image](#) now also supported the ARM64 architecture and allows to run all the CLI commands [#532](#)
- Support for Snowflake added [#520](#)
- Support for MariaDB added [#271](#)
- N-Triples and N-quads are now supported by the SPARQL endpoint [#566](#)
- Option to fully reformulate the SPARQL query into SQL added [#577](#)
- Experimental support for flatten lenses added (only for PostgreSQL)

New optimizations

- Optimization of joins with a union of class definitions added [#543](#)
- Optimizations for generic queries with limit added [#578](#)
- Optimization for queries retrieving all the properties or classes in use in the VKG added [#581](#)

Refactoring

- New module for testing dialects in the Github CI pipeline added [#568](#)
- JSQParser updated to 4.4
- Ontop's Protégé plugin isolated from Ontop modules [#278](#)
- Support for Databricks (Apache Spark) improved

Many bugfixes. See also [its milestone on Github](#).

4.2.2 (November 18, 2022)

New features

Support for Protege 5.6-beta under Java 11 and Java 17

- Support for the Databricks-specific JDBC driver [#554](#)
- Support for LIMIT/OFFSET in SQL Server 2008 [#531](#)
- Support for materialized views in PostgreSQL [#541](#)

Bugfixes

- VALUES handling with Teiid [#525](#)
- Metadata extraction for Dremio of schema names with dots
- Typo in registration of Oracle factories [#557](#)

Preventive update of dependencies including Jackson and Gson to address potential vulnerabilities.

4.2.1 (April 20, 2022)

New feature

- Support for partial serialized DB metadata, when `ontop.allowRetrievingBlackBoxViewMetadataFromDB` is enabled.

Bugfixes

- Not renamed variable in sub-query with a distinct ([#417](#)).
- OBDA to R2RML conversion: require mapping entry IDs to be unique.
- Lang tags for constants in R2RML were ignored.
- Several fixes related to VALUES.
- Better HTTP codes for the ontology download endpoint.

The Jackson and Spring Boot dependencies have been preventively updated (to 2.13.2.2 and 2.6.6 respectively) to address potential vulnerabilities ([CVE-2020-36518](#) and [CVE-2022-22965](#)).

4.2.0 (December 30, 2021)

New features

- Support for Apache Spark ([#422](#)).

Support for time functions added ([#478](#)).

- Support for the `IN` SPARQL function added.
- Datatypes can be extracted from non-supported source queries in the mapping (treated internally as "black-box" views). Disabled by default (see `ontop.allowRetrievingBlackBoxViewMetadataFromDB`).
- TBox triples (e.g. sub-classes, domains and ranges) can be added to the default RDF graph. Disabled by default (see `ontop.enableFactExtractionWithTBox`).
- Support for [Ontop join views](#) added.
- Support for arbitrary levels of Ontop views added (views over views).
- Better integrity constraint extraction for Ontop basic and join views.
- Non-null information can now be specified for Ontop views.
- Support for b-node facts added.
- The default query of the SPARQL endpoint portal is now configurable ([#454](#)).
- Contextually injective IRI templates with numeric or UUID arguments are now decomposed ([#363](#)).
- JSON-LD serialization now supported by the materializer ([#481](#)).

Refactoring

- Support for Java ≥ 8 . Ontop decoupled from Protégé so as to update several common libraries ([#467](#)).
- RDF facts provided by the ontology file are now internally grouped together into values blocks (sorts of "in-memory tables") instead of being splitted into large unions. Enabled by default (see `ontop.enableValuesNode`).
- Unions are lifted above aggregations when possible.
- Docker image updated to JRE 11.
- Mutable intermediate queries are not used anymore.
- Stronger handling of non well-designed `OPTIONAL` blocks.
- Stronger handling of `ORDER BY` in sub-queries.
- Type extractor relaxed ([#438](#)) to tolerate slightly different datatypes.
- JDBC distinct result set removed.
- JDBC user name and password are not required anymore. Useful when passed in the JDBC URL.

- The Command-Line Interface of the bootstrapper has been refactored (invalid options removed, comments improved).

Many bugfixes (see for instance [our Github bugtracker](#)).

4.1.1 (July 23, 2021)

Bugfixes

- Bugs related to GeoSPARQL support ([#428](#) ,[#434](#))
- Issue with ORDER BY and LIMIT for MS SQL ([#225](#) ,[#412](#))

New features

- Teiid support ([#336](#))
- Various improvements of the Protégé plugin

4.1.0 (February 28, 2021)

Changed behavior

- [OBDA to R2RML mapping conversion](#) now requires by default access to DB metadata (through credentials or as a serialized file). This requirement can be bypassed with the argument `--force` .
- By default, DESCRIBE queries only return the triples where the described IRI appears as subject. To also consider the triples where it appears as object, set the parameter `ontop.includeFixedObjectPositionInDescribe` to true.
- The Docker endpoint image is now based on Debian ([#394](#)), not on Alpine anymore.

New features

- Support for GeoSPARQL functions added ([#335](#)).
- [Rich JSON-based query logging](#) added.
- [New endpoint for predefined queries](#) added (beta). It enables the specification of custom low-latency Web APIs in a declarative manner.
- Support for the `IF` , `IRI` , `BNODE` SPARQL functions added.
- Support for Dremio added.

- DB metadata can now be loaded from files instead of connecting to the database.
- Bnode labels are now anonymized on-the-fly. Bnode templates can safely use PII.
- Sensitive JDBC information (user name, password, URL) can now be passed as arguments or environment variables (for Docker) instead of being written in the properties file. Docker secrets are also supported for further security.
- New left join optimization techniques added. General functional dependencies are now taken into account. Sensitivity to left join ordering reduced.
- [HTTP caching](#) headers can now be returned by the Ontop SPARQL endpoint.
- Native Protégé bundles for each platform (Windows, MacOS, Linux) with JRE 8 embedded.
- Basic support for Ontop views added (experimental), which, in particular, allows specifying integrity constraints on views defined at the Ontop level.

Removal

- As announced earlier, the RDF4J Workbench-based bundles are not shipped anymore. However, the webapps war file can still be built with Maven.

Refactoring

- Drastic reduction of memory consumption when processing SPARQL queries ([#370](#)), which is significant for materializing large RDF graphs.
- Better file resource handling ([#368](#)).
- Distinct lifting improved.
- Self-join elimination based on functional dependencies re-implemented.
- Decomposition of heterogeneous IRI templates enabled.
- Lexical value space is now considered when decomposing IRIs.
- Protégé plugin codebase cleaned, with parts of UI streamlined and redesigned. Improved syntax highlighting in target and source queries. Mapping validation extended. Multiple open ontologies supported ([#282](#) and [#175](#)).

- RDF4J external binding management refactored.
- Bnode handling refactored.
- Java 11 support.
- Better handling of column names when converting OBDA mappings into R2RML, which now uses DB metadata.
- CONSTRUCT and DESCRIBE query handling refactored.
- Parser for OBDA mappings refactored.
- Character encoding in R2RML templates refactored.
- Default file encoding set to UTF-8. Particularly useful on Windows. Can be overridden if needed.
- [Build script](#) refactored. Now fully based on Maven ([#393](#)).
- Base Docker image updated.
- PostgreSQL transaction handling (in streaming mode) improved.
- Timeout support extended to non-SELECT queries.

Many bugfixes.

4.0.3 (October 16, 2020)

Bugfixes

- Wrong IRI decomposition ([#358](#))
- Logback config file ignored by Ontop CLI ([#361](#))

4.0.2 (September 3, 2020)

- Fixed Named Graphs (quad) grammar in the target of the Ontop native mapping language.

4.0.1 (August 28, 2020)

New features and bugfixes

- Named Graphs (quad) are supported in materialization ([#343](#))
- Removal of Jena as a dependency ([#349](#))
- `com.mysql.cj.jdbc.Driver` is now also recognized. Non-recognized drivers now use the default SQL factories instead of the mock-up one.

4.0.0 (July 31, 2020)

Refactoring

- Dependencies updated (e.g. RDF4J 3.3.0)
- SERIAL datatypes of PostgreSQL better handled
- Anonymous blank nodes are now supported in CONSTRUCT blocks
- JSON-LD now supported for graph queries
- Denodo support improved.

Few bugfixes.

4.0.0-rc-1 (July 8, 2020)

Changed behavior

- Streaming mode is enabled by default for query answering. It fetches data by batches of 500 results.

New features

- Support of GROUP_CONCAT added. All the standard SPARQL aggregation functions are now supported.
- Named graphs are now supported.
- Support of Denodo added. Support of Dremio is postponed to 4.1.
- Support XML catalog file added (beta). It enables resolving `owl:imports` assertions from the ontology by loading local files instead of fetching remote URLs. It is available in Protégé and with the `ontop endpoint` CLI command.
- DB metadata extraction CLI command added (experimental).
- CASE and CAST is now supported in the mapping.
- Streaming mode for query answering added.

Deprecation

- The RDF4J Workbench-based SPARQL endpoint is deprecated. Tomcat, Jetty bundles and webapps (.war files) will be removed in a future version. Users are recommended to switch to the `ontop`

`endpoint` CLI or its Docker image.

Removal

- The Mapping Assistant panel has been removed from the Ontop Protégé plugin.

Refactoring

- SQL parsing of the mapping improved. JSQLParser's version has been updated.
- Metadata extraction refactored, and SQLDialectAdapter has been eliminated.
- Meta-mapping management improved.
- Non-recognized SQL functions in the mapping are now only assumed to be deterministic. Other assumptions have been removed.
- Internal data nodes now create variables only for database columns that are used in the query. This sparse usage of variables strongly improve reformulation time and log readability when using tables with 100+ columns in the mapping.

Many bugfixes.

4.0.0-beta-1 (December 23, 2019)

Changed behaviors

- Cardinality is now enforced by default (duplicates are removed at the mapping level). This is required to support most of the aggregation functions.
- Functions in the SQL queries of the mapping that are unknown to Ontop do no cause the creation of SQL views anymore. The strategy regarding SQL functions moved from white-list to black-list.
- Types returned by supported SPARQL functions now strictly follow the SPARQL 1.1 specification.

New features

SPARQL Aggregation functions SUM, COUNT, AVG, MIN, MAX are now supported. SAMPLE is currently implemented as MIN.

- Joins over terms that are constructed using different functions are now supported.
- Some functions (such as IRI templates) can now be post-processed (that is, they are not always included in the SQL query anymore). Generated SQL queries are generally less verbose than before.
- The Ontop endpoint now has the portal option, where predefined queries are shown in the Web interface.
- The NULLIF SQL function is now supported and optimized. Useful for filtering out some unwanted values in the mapping assertions.
- OPTIONALs outside the well-designed fragment are now supported. Weakly well-designed queries encoding preferences are optimized.

Refactoring

- The typing system have been entirely redesigned. RDF types and database types are now clearly separated. Each database engine can now bring in its own data types.
- The Datalog data structure (from version 1.x) for representing queries internally has been removed.
- Function symbol creation and optimization has been entirely refactored. They are constructed by two factories, the FunctionSymbolFactory and the DBFunctionSymbolFactory.
- Most of the logic of SQLDialectAdapter has been moved to DBFunctionSymbolFactory and DBTypeFactory. Each database engine is expected to have its own implementation of these new interfaces.
- The SPARQL translation has been rewritten (was formerly translating into Datalog).
- The SQL generator has been rewritten (was formerly translating from Datalog).

Temporary limitation

- Due to a complete refactoring of the SQL dialect management, only the H2, MySQL, PostgreSQL, Oracle, SQL server and DB2 dialects are supported in this beta. Dremio and Denodo dialects

are expected to be supported in a next beta release.

Many bugfixes.

3.0.1 (December 9, 2019)

Changed behaviors

- Protégé: no property file (resp. mapping file) is created if no data source (resp. mapping assertion) is specified by the user.

New features

- Ontop endpoint now supports R2RML files and doesn't require an ontology.
- Materialization: N-triples format is now supported.

Refactoring

- Rewriting of the tree-witness rewriter (now independent from Datalog)

Bug fixes

- Protégé: fixed issues related to prefix management.
- Reasoning: issue related to the handling of inverse properties fixed.

3.0.0 (August 23, 2019)

Changed behaviors

- The ``rdfs:range`` of a data property is not used anymore for inferring the non-specified datatype in a mapping assertion but for validating the datatype once specified or inferred. Datatype inference is now based on the DB datatype and follows the R2RML natural mapping (<https://www.w3.org/TR/r2rml/#natural-mapping>)

New features

- Support of Protege 5.5
- Support of BNode in R2RML
- Basic support for the `geo:wktLiteral` datatype (#256)
- New command-line interface for SPARQL endpoint
- Docker container for SPARQL endpoint
(<https://hub.docker.com/r/ontop/ontop-endpoint>)
- Preliminary support of Dremio
- Preliminary support of Denodo
- The ordering of SPARQL variables is now preserved (#291)
- N-Triples output for materialization added
- The SQL translation now appears as a sub-tab in the Protégé Ontop SPARQL tab

Refactoring

- The triple predicate is now systematically used in the mapping and during query unfolding
- Atom predicates and function symbols are now clearly distinguished
- The Datalog-based data structure is not used anymore at mapping processing time and by the default rewriter
- Query containment-based optimizations applied to the mapping after saturation improved
- The materializer is now based on RDF4J (instead of OWLAPI)

Project organisation

- Release of Ontop on both SourceForge and GitHub (#274)
- A new repository for the tutorial
(<https://github.com/ontop/ontop-tutorial>)

Many bugfixes.

3.0.0-beta-2 (February 28, 2018)

Changed Behaviours

- The JDBC driver class name is now required in Protégé (issue #248)

New features

- Novel left join optimization techniques (issue #230)
- LIMIT appended to test SQL queries and SPARQL queries in Protégé
- Report illegal usage of Fully qualified column names (FQDN) in the target part of a mapping assertion
- Support for the SPARQL isNumeric function

Refactoring

- Internal representation of the ontology improved
- Improvement of the intermediate query representation (issues #235, #236)
- Switch to ANTLR4 to parse ontop native mapping (issues #238, #239)
- Make ResultSet.next() robust (issue #219)
- More robust handling of IRI-safe encoding (issue #245)

Bug Fixes

Many, see [closed issues](#) .

3.0.0-beta-1 (August 11, 2017)

Note that this is the first release from version 3 series with many internal refactoring and improvements, and bug fixes. However, these changes also cause incompatibility with Ontop v1. See [the migration guide](#).

Changed Behaviours

- JDBC connection information is stored in a separate properties file
- Fully qualified columns (table.column) are not allowed in the target part of the mapping language. One has to explicitly create an alias (e.g. table.column AS table_column) and use the alias in the target. (complying with R2RML)
- The rules for datatype type inference and checking are changed. Now we use only mapping and database to infer the datatypes (complying with R2RML), and the (range axioms) in the ontologies

to check the compatibility (complying with OWL).

New features

- Major refactoring of the internal structure of the Ontop system
- Databases for Integration Tests have been dockerized.
- Upgrade Sesame to the latest version of RDF4J (v2.2.2).
- The redesigned OWLAPI and RDF4J API bindings of Ontop API (see examples: <https://github.com/ontop/ontop-api-examples>)
- Upgrade of R2RML API to v0.6.0
- Official support of both Tomcat and Jetty (v9) for deploying a SPARQL endpoint
- Better support of Optional (bugfix #190)
- Protege: support for "Synchronize Reasoner" in the Reasoner menu
- Improved error messages for incorrect mapping
- CLI tool for converting Mapping from v1 to v3

1.18.1 (June 16, 2017)

- Return a correct extension for files in ontop cli materializer (issue #145)
- Materialize support non-http resource uris
- Bug Fixes
 - Small fixes in cli (issues #127, #141, #188)
 - PostgreSQL JDBC driver fix for quotation marks
 - Remove unknown property warning for owl:sameAs
 - Remove duplicate annotations during materialization (issues #183 #186)
 - Generate different file names for properties and classes during materialization with separate files (issue #165)
 - R2RML
 - Small changes in api
 - Fix R2RML import of SQL query (issues #181 #182)
 - Fix export from OBDA mapping to R2RML (issues #158 #159 #160)
 - Fix in SPARQL use of filter and values with a IRI (issue #191)

- Fix handling of mysql metadata with lower_case_table_names setting on (issue #192)

1.18.0 (May 26, 2016)

- Protege
 - Redesigned materialization gui
 - Upgrade to Protege 5.0.0
- Support of querying annotation property from ontology (issue #105) (disabled by default)
- Support of the use of owl:sameAs in mappings (disabled by default)
- Package names have been unified to it.unibz.inf.ontop
- Upgrade OWLAPI to v4.2.5
- Bug Fixes
 - Fix obda-to-r2rml convert for template like <{uri}> (issue #97)
 - Fix an issue in Protege when renaming data source.
 - Fix error messages in protege mapping assistant panel and SPARQL query
 - Fix issue with multiple schemas (issue #99)
 - Fix internal SQL conversion for Teiid jdbc driver
 - Fix support of anonymous ontology (issue #101)
 - Fix warning message in Protege (issue #102)
 - Fix xml uri (issue #104)
 - Fix r2rml api, error predicateMap with template (issue #107)
 - Fix in Protege import of additional ontology and add classes and properties of imported ontologies at runtime (issues #106 #108)
 - Fix querying URI constants (issue #111)
 - Fix error message creating new mapping (issue #120)

1.17.1 (February 16, 2016)

- Bug Fixes
 - Fix an issue when creating new datasource in Protege

1.17.0 (February 4, 2016)

- Protege
 - Ontop protege plugin now available on official Protege Plugin Repository!
 - Upgrade Protege to 5.0.0-beta-21 (latest version)
 - Ontop protege plugin is packed as one jar now -- no need for separated OSGI JDBC configuration jars
 - do not generate .obda and .q file when no datasource is present
 - General improvement of UI -- more intuitive and less confusion
- Support for Annotation Properties in mappings
- Improvement of Ontop mapping language for URIs/Literals retrieved from DB columns
- Support for SPARQL VALUES keyword for SPARQL federation
- Java 8
- Upgrade OWLAPI to v4.1.3 (latest version)
- Bash Completion for command line interface
- Better structured Wiki: <https://github.com/ontop/ontop/wiki>

1.16.1 (November 16, 2015)

- Optimizations exploiting foreign key constraints over multiple columns
- Improvement in the RDB2RDF compliance test (contributed by Evren Srin)

1.16.0 (October 14, 2015)

- Support for [SPARQL functions](#)
- Support for column oriented RDBMS [MonetDB](#)
- Support for cloud RDBMS [SAP HANA](#)
- New Optimization technique: [Implicit-database-constraints](#)
- New Optimization technique: [Exact-Mappings](#)
- Bug Fixes
 - Handling quotes in the WHERE clause of SQL query in mappings .
 - Allow multiple subjects in the target of mapping of Ontop native syntax.
 - Return correct value from datetime columns using jtds driver
 - Fix the translation of LIMIT with ORDER BY in oracle database

- Fix a bug in MetaMappingExpander

1.15.0 (May 13, 2015)

- Upgrade the Protege plugin to Protégé 5
- New Command line interface
- Better support for SPARQL BIND
- Support for SPARQL Concat and Replace
- Support for SQL Concat and Replace (in the mappings)
- Support for xsd:dateTimeStamp
- Improved Optimization techniques (Contribution by Roman Konchakov)
- Implementation of Autoclosable interface for core classes in the API to support `try with resource``.
- Create a sub-query when SELECT DISTINCT is used in the mappings.
- Bug Fixes
 - Handle the 30-characters limit for variable names in Oracle.
 - Fix the mapping render of Protege when multiple subjects are in the target of mapping.

1.14.0 (November 4, 2014)

- Added support in ontology and mappings for datatypes xsd:int, xsd:long, xsd:positiveInteger, xsd:negativeInteger, xsd:nonPositiveInteger, xsd:nonNegativeInteger, xsd:float
- Optimized implementations of TBox classifications, and query containment checking, which are much faster than before
- Validation for Sesame Workbench made stronger. It prevents repositories with an invalid configuration to be created.
- Bug Fixes:
 - Fix SPARQL functions (str(), lang()) to use rdfs:Literal instead of xsd:string
 - Issue converting datatypes of R2RML constants
 - Support UNICODE characters in TurtleParser
 - Catch exceptions for Protege R2RML Export
 - Fix conversion from ontop mappings to R2RML of language tag and triples

1.13.1 (September 29, 2014)

- Bug Fixes:
 - The redundant query elimination optimization sometimes does not handle facts properly
 - Some warning messages related to datatype range axioms are ambiguous

1.13.0 (September 25, 2014)

- Support for providing implicit DB Constraints (e.g. foreign keys) to Ontop to generate efficient SQL
- Command line feature to generate pretty R2RML
- Stream output in the Protege Plugin
- Support for Regex in mappings
- Proper handling of datatypes in ontology and mappings
- Support for hsql database
- Support for datatypes xsd:date, xsd:time and xsd:gYear
- Bug Fixes:
 - Issue with nested equivalences in unfold
 - Problem with multiple join conditions in R2RML
 - Cancelling running queries in Protege
 - Blank lines in generated sql causes problems for oracle

1.12.0 (June 26, 2014)

- Support for finding empty concepts and properties
- Integration with a new R2RML API
- Less restrictive mode for JDBC driver validation
- Faster TBox reasoner implementation
- Optional "Replace" Statement in the SQL translation. When "Replace" is off the SQL performs much better.
- Several Bug Fixes:
 - Issue with 'true' in Oracle
 - Meta-mapping editing in Protege
 - Problem with regex in Oracle
 - Several issues with R2RML

1.11.0 (February 19, 2014)

- Support for Consistency Checking
- Improved Performance
- Support for Multi-schema queries
- New SQL parser for the mapping based on JSQLParser.
- Extended the supported SQL language: IN, BETWEEN, LIKE, etc.
- Extended the supported SPARQL: BIND
- Improved the Treewitness rewriting algorithm.
- Upgraded to Sesame 2.7.10
- General clean-up
- Fixed a number of bugs:
 - Translation of SAPRQL REGEX into SQL
 - Infinite loop triggered by some nested Optionals
 - Several issues with Construct
 - Several issues with R2RML

1.10.0 (December 16, 2013)

- Ontop under Apache license
- Ontop packages in central maven repository

1.9.0 (September 18, 2013)

- Update to Sesame 2.7.6 (latest)
- Validation of datatypes w.r.t. XMLSchema specification
- New hybrid Abox functionality support
- Removed all dependencies to Jena
- Removed wrong/unnecessary dependencies
- Turtle parser can handle variables in the predicate
- Mappings with URI templates in class or properties support
- First version of mapping splitter
- Bug fix: Null pointer exception when the URI in Protege plugin was empty
- Bug fix: Literal and literal with language tag handling
- Bug fix: Shutdown of Sesame repoConnection
- Bug fix: Object column literal generation in R2RML
- Bug fix: Queries that would not finish execution

- Bug fix: Unexpected bindings in unfold
- Bug fix: Null pointer exception in ontology translation in the case of unsupported axiom
- Bug fix: OWL class assertion generation, inverted arguments
- Bug fix: SameTerm in sparql algebra to datalog translation
- Bug fix: URI template matcher to find most specific matches first
- Bug fix: Construct queries
- Bug fix: Describe queries

1.8.0 (July 29, 2013)

- Improve error message and better exception handling.
- Change the representation of `null` value in BindingSet
- Modified the computation of the LeftJoin to support empty evaluation in its second argument.
- Changed the type of the semantic index table for integer data property assertion from INT to BIGINT to be able to handle `xsd:integer` type
- Set index creation during Quest startup
- Implement new feature for bootstrapping, command line scripts, integration in ontoPro plugin
- Updated the URI templates in Turtle mapping syntax to make it simpler and closer to Turtle. Now quotes, question marks and class/property declarations are no longer required

For example, in the previous syntax:

```
<"test;person/{?id}"> a :MyClass; :hasName {?name}.
```

becomes simply

```
test:person/{id} a :MyClass ; :hasName {name} .
```

- Support "%", "=", ";" characters in the URI template.

```
:class/{id};{name}%{class type}
```

- Bug fix: Major bugs in execution handling SPARQL Construct and

SPARQL Describe queries.

- Bug fix: Issues related to commits and batch sizes in QuestOWLStatement
- Bug fix: Issue with URI template matcher that generates `null` in case failed template (now it generates `URI("xxx")` where xxx is the URI that was being matched.)
- Bug fix: Issue in existential clause in tree-witness algorithm
- Bug fix: Issue in the maximum size of URI that has more than 400 characters
- Bug fix: Issue in some data that can not be tracked when inserted into the database. In particular if the data comes from classes/properties that were simplified by the equivalence resolution.
- Bug fix: Issue in exception handling during index creation in semantic index test.
- Bug fix: Issue in SQL parser that it doesn't capture the scope within parentheses in the WHERE clause.
- Remove table and column identifier quotes in the SQL target query string so that the parser can parse properly. For example:

```
SELECT "Student"."sid" from "Student"
```

will become:

```
SELECT Student.sid from Student
```

- New semantic index implementation with integer references for URIs for faster query answering (faster joins on integers instead of strings)

Ontop source code

- Ontop is now in GitHub!
- Cleaner and slimmer project structure. We removed obsolete classes and tests.
- Code refactor in Materialiser API, MinimalCQProducer and Construct-Describe query execution.
- More complete test cases for testing Quest features, e.g., Tree-Witness rewriting, Data materialiser, SQL parser, SPARQL

Construct, SPARQL Describe, SPARQL Optional, Database metadata and Query completeness.

- Support JUnit4 test cases.

1.7-alpha2 (rev 2305, April 5, 2013)

- Support for importing/exporting R2RML mappings
- Support for automated mapping generation as defined by the "Direct Mapping" W3C specification
- Various options for data materialization (similar to dump-rdf in D2RQ)
- Support for concurrent queries in the SPARQL endpoint through connection pooling
- Better support for Oracle, Postgres, SQL Server and DB2
- Improved the SQL queries generated by Quest by avoiding some operations when unnecessary (e.g., casting), and placing optimally the JOIN conditions in the SQL query.

1.7-alpha2 (rev 2178, February 7, 2013)

-ontopQuest-

- Mapping optimiser: improved the generation of OR mappings
- Mapping optimiser: improved the simplification of mappings using FOREIGN KEYS
- Query optimiser: Improved the query optimiser to avoid generation of redundant JOINS and redundant UNIONS w.r.t. FOREIGN KEYS
- Support percent encoding in strings in URI templates as defined by the R2RML standard.
- Better propagation of Exception messages in the Sesame provider
- New scripts for querying quest from the command line

-ontopPro- plugin

- Introducing mapping assistant. A new GUI-based tool for creating mappings. This is an experimental GUI to help you create mappings in a faster way, please help us improve it by reporting any issues you may find. (See [guidelines](#)).
- Users can set the max number of tuples returned in "OBDA

Query" tab.

Mapping language features (-ontopQuest- and -ontopCore-)

- Support URI identifier obtained directly from the database (no using URI template).
- Support multiple lines when wrting SQL source query (See [feature notes](#))

Bug fixes

- Bug fix: Issue with MySQL and UTF8 characters (Korean in particular)
- Bug fix: Issue with "unbound variable freshX_X found while generating SQL" fixed.
- Bug fix: Inverse minimum cardinality assertion was being ignored.
- Bug fix: Source SQL query doesn't accept parenthesis in WHERE clause.
- Bug fix: JDBC connection doesn't recover when connection lost.
- Bug fix: Sesame virtual repository fails to read Quest preferences.
- Bug fix: CSV export function only exports 101 tuples instead of all the tuples.
- Bug fix: Null pointer exception in the query result table.

1.7-alpha (rev 2039, December 30, 2012)

SPARQL support

- Support for most SPARQL built in functions, e.g., arithmetics, type checking, etc.
- Support for CONSTRUCT and DESCRIBE result forms
- Support for OPTIONAL and bound, which allow you to have NEGATION in the usual SPARQL 1.0 style.
- Support for variables in ANY position, e.g., to allow queries such as SELECT ?p ?v WHERE { ?x :name "John Smith". ?x ?p ?v }
- Stronger compliance to standard RDF and SPARQL semantics for FILTER expressions

Performance

- Multiple improvements to the SPARQL-to-rule translation, mapping and TBox optimization algorithms.

- New algorithm for query rewriting w.r.t. the TBox that increases the performance of query answering by many orders of magnitude when inference w.r.t. to existential constants is enabled. The implementation has been contributed by Dr. Roman Kontchakov.
- New benchmarks. We tested against Virtuoso triple store, Virtuoso RDF views and D2R Server using the BSBM and FishMark benchmarks.

APIs and Accessibility

- Completely new Sesame API provider, now users of Sesame can create "Virtual RDF repositories" with -ontop-
- SPARQL end-point support through integration with Sesame's workbench

Mapping language

- Improved support for default literal types, now it conforms with the R2RML "Natural Mapping" specification
- Support for language tags obtained from the DB
- Proper support for constants in the mappings
- Proper internal handling of NULL values in the database (as recommended by the R2RML standard)

Database support

- Restored support for Oracle, DB2 and SQL Server
- Improved support and performance over MySQL

Documentation

- Revised and extended all technical documentation

1.7-alpha-rev-1692 (August 16, 2012)

-ontop- general

- New file content for the OBDA output file. The file is no longer in XML format but instead it uses plain text for simplicity (Issue: #244).
- Improved error messages when the system initializes, parses mappings, fails to

`execute queries, etc.`

-ontopPro- Protege plugin

- Support Protégé 4.2
- Add cancel button during query execution to terminate the process immediately.
- Add help button to guide filling data source parameters.
- The plugin no longer introduces prefix definitions
- Implemented "Cancel" function for query execution
- New shortcut CONTROL + T to test mappings in the mapping creator
- Fixed several issues with prefix management and rendering of

`URIs short forms`

- Improved GUI, hints and suggestions for JDBC connections, improved TAB navigation

`in all dialogs`

- Bug fix: Improve error reporting and stack trace printing (Issue: #255, #262)
- Bug fix: The target and source string got saved partially if newlines are used.
- Bug fix: The OBDA model doesn't get loaded if the query has an exception.
- Improved error management during .obda and query file load
- Fixed null pointer exception and blank window that appear in cases where the .owl file

`in OWL/RDF contained different default ns and a default prefix`

-ontopQuest- OWL Reasoner

- Added support for SPARQL query modifiers: ORDER BY, OFFSET and LIMIT (Issue: #261).
- Add query caching to speed up the system performance, now repeated query execution will not require SQL generation every time.
- Relaxed on punning checks to allow loading of DBPedia and

similar ontologies.

- EXPERIMENTAL: Added support for mixed ABoxes for class assertions, i.e., most data comes from the mappings but a bit of data is also allowed in the ABox, e.g., global facts Planet('mars')
- Improved support for MySQL, Postgres, DB2 and SQL Server dialects
- Added a warnings when axioms are being ignored because of being out of OWL 2 QL
- Improved error reporting on failed connections
- Improved support for SQL AS for table and column renaming in mappings
- New way to generate SQL. SQL now includes typing information and is pretty printed.
- Bug fix: Fixed issue with INVALID URI that appeared when mappings used data that contained invalid characters for URI's (e.g., white space) Now we call URLEncoder (Jena's) before returning the value.
- Bug fix: Exception when importing data as ABox individuals.
- Bug fix: Exception when null values is used as constants.
- Bug fix: Exception if ambiguous column names are used in SQL queries.
- Bug fix: Wrong URI shorten if there are two or more namespaces with overlapping strings.
- Bug fix: Null pointer exception due to inconsistent OBDA model identifier string.
- Bug fix: Unknown signature in the SELECT clause if the name is a qualified name and it comes from a view (Issue: #259)
- Bug fix: Exception when using URIs as a constant values in SPARQL.
- Bug fix: Prefix Manager accepts any string including invalid prefix names.

Known issues:

- The ORDER BY modifier is not ready for columns with literal datatype
- Issues with MySQL in non-ANSI mode
- DateTime data properties in Oracle can crash the system.

1.7-alpha-rev-1529 (June 12, 2012)

-ontopPro- plugin

- Better layout, fonts and command icons.
- Better query saving mechanism (Issue: #248)
- Better error reporting to users and crash handling (Issue: #255, 256)
- Pretty printing of SQL queries (Issue: #254)
- Re-enabled the export function in the Query Interface panel (Issue: #213, #250)
- Improved syntax-error messages in the mapping editor tooltip
- Cleaned console messages
- Added "Number of sources" indicator to show how many sources are already declared in the OBDA model.
- Improved ESC and CONTROL+ENTER shortcuts in most dialogs in the GUI.
- Replaced JDBC Driver text box with an editable Combo Box with default driver string.
- Better OBDA statistics table (Issue: #251)
- Bug fix: CRITICAL stability issue that would rise an end-less cycle of warning messages when Quest crashed.
- Bug fix: issue with final dot "." in mappings.

-ontopQuest-

- Added support for data-type UUIDs for SQL Server (Issue: #253)
- Now we create forward-only and concur-read-only JDBC Statements by default to allow for server side cursors.
- Bug fix: issue with SQL parser and table renamings in SQL queries (Issue: #259)
- Bug fix: issue with NULL pointer when primary key optimization in unfolding when there is a join on primary condition but atoms are not unifiable.

Known issues:

- Oracle is unstable

1.7-alpha-rev-1479 (May 22, 2012)

Quest performance and features:

- T-Mappings algorithm optimization: Combine disjunctive conditions in OR clauses instead of multiple mappings. This reduces the size of the final query and improves its performance. (Issue: #237)

Mapping language features (Quest and ontopCore):

- Support using IS NULL and IS NOT NULL in SQL query (Issue: #200).
- Support using UNION in SPARQL queries.

Bug Fixes:

- Fix packing problem in Windows XP (Issue: #228)
- Fix packing problem in Linux
- Fix unresponsive plugin when using mapping filter (Issue: #217).
- Fix unresponsive plugin when inserting a new mapping (Issue: #232).
- Fix support for MySQL in both virtual and classic mode (Issue: #238).
- Fix support for Oracle in virtual mode (Issue: #226).
- Fix issue where creating or modifying a mapping would not trigger a "ontology out of synch" flag.

Known Issue:

- Oracle still has some issues in classic mode.

1.7-alpha-rev-1446 (May 4, 2012)

Mapping language features (Quest and ontopCore):

- Add support on writing qualified name for atom predicates.
- Add coloring for atoms typed Individual in the target query.

Bug Fixes:

- Fix the "Show Short URIs" feature in the query panel.

1.7-alpha-rev-1440 (May 2, 2012)

Mapping language features (Quest and ontopCore):

- Support writing URI constants in target queries (Issue: #227)

General:

- Better system logging for recording exceptions.

Bug Fixes:

- Fix the null pointer exception due to the old OBDA preferences.
- Fix bugs related to vocabulary prefixes.

1.7-alpha-rev-1427 (April 17, 2012)

Quest performance and features:

- Implemented the T-Mappings mapping optimization algorithm improved performance of query rewriting in Virtual Mode. Now query rewriting is as fast in virtual ABox mode as in classic ABox mode (Issue: #169)
- Implemented a new query containment algorithm that is many times faster than before.
- Added data source analysis to obtain DB metadata for SQL query generation.
- Added analysis of SQL queries in the mappings to produce better SQL queries.
- Added SQL JOIN ELIMINATION using PRIMARY KEY metadata.
- Added mappings optimization using query containment w.r.t. constraints
- Added initial support for FILTER
- Added support for typed constants in SPARQL queries (Issue: #157)
- Small updates to internal API for speed of equality, hash and clone computation

Mapping language features (Quest and ontopCore):

- New Turtle-like syntax for mappings allows for more succinct and easy to read mappings.
- Added support for LANGUAGE tag for the Literal datatype (Issue: #198)

- Added support for datatypes in the OBDA mapping axioms (Issue: #147, #194, #196)
- Added URI templates for URI creation in mapping axioms (Issue: #151)
- New feature for typing literals in mapping axioms (Issue: #194, #207)

General:

- Added support for OWL API 3 (Issue: #98)
- Internal refactoring of all modules and packages for a cleaner API

ontopPro:

- The plugin is now compatible with Protege 4.1 (Issue: #99)
- The plugin has a new and fresh GUI display.
- Stronger integration with Protege's refactoring mechanism
- The ABox materializer can import data complete with datatype information (Issue: #195)
- The target query use Turtle syntax for better display and management (Issue: #204)
- The plugin has Class/Property panel for better navigation and filtering.

Bug Fixes:

- Fixed a problem with the encoding of axioms of the form `subClassOf(Person, hasFather some Person)`
- Fixed issues with open and dead JDBC connections
- Cleaned OWLReasoner synchronization issue
- Fixed null pointer exceptions related to incorrect mappings
- Proper error handling for translating an empty datalog program (Issue: #197)
- Fixed the "Attach Prefixes" feature in the OBDA Query panel (Issue: #211)
- Fixed issue regarding `xsd:dateTime` (Issue: #199)

1.6.1 (October 19, 2011)

- Fast patch in the query translation module.

1.6 (October 18, 2011)

- Allow fast navigation using TAB key in the new mapping form (Issue: #92)
- Allow SQL query execution in the mapping tree (Issue: #104)
- Improvement in the error message deliveries (Issues: #107, #112, #118)
- Improvement in the data source management tab, including adding "Test Connection" button (Issues: #108, #111, #114)
- Add OBDA model statistics report (Issues: #106, #117)
- Add support for adding integrated data source in Teiid (Issue: #89)
- Fix the unresponsive cancel button during importing OWL Individuals (Issue: #88)
- Fix CSV header in the exported query result (Issue: #120)
- Fix connection failure when users change the parameters (Issue: #121)
- Fix the password field so now the text is hidden (Issue: #113)
- Fix query selection in the OBDA query tab (Issue: #91)
- Fix import failure when storing very long record values (Issue: #95)
- Fix import failure when storing records that contain single quotes (Issue: #96)

1.5.1-RC1 (July 15, 2011)

- Significant code refactoring in the plugin. This effort makes the code easier to debug and to extend, and is almost ready for Protege 4.1 and OWL-API 3.0 support.
- Term renaming. Modified the term 'Mapping Head' to 'Target Query' and 'Mapping Body' to 'Source Query' to maintain the consistency with the literature.
- Temporarily disabled the SPARQL query coloring.
- Temporarily disabled the 'Load OWL Individuals into DB' functionality.
- Add improved search filters in the mapping manager (Issues: #37, #54).
- Add 'show short URIs' option in the query results to hide long URI strings.
- Add 'select all', 'select none', 'expand all' and 'collapse all' buttons

for easy browsing in the mapping tree (Issues: #35, #36).

- Add mapping validation with respect to the vocabulary in the ontology (Issues: #42).
- Add input validation to many parts of the GUI (Issues: #63, #64, #65, #67, #68, #69, #70, #71, #76).
- Fix the save query functionality in the Query manager (Issues: #57).
- Fix the persistency of user preferences when opening/closing Protege (Issues: #58).
- Fix the exceptions that appeared when loading/saving/reloading ontologies (Issues: #59, #60, #61, #75).
- Fix problems with several progress dialogs in the GUI (Issues: #50, #87).
- Fix problems in the 'tuple count' functionality (Issues: #82, #84).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Glossary](#)



Ontop

A Virtual Knowledge Graph System

[Get Started →](#)

Documentation under CC-BY license | Copyright 2009-present - Free University of Bozen-Bolzano

Tutorial

Presentation

Requirements

Clone this repository

Program

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Presentation

In this tutorial, we will see how to design a Virtual Knowledge Graph (VKG) specification, how to deploy it as a SPARQL endpoint, how to consume it and further more advanced topics.

Requirements

- [Java 11](#)
- Latest version of Ontop from [GitHub](#) or [SourceForge](#)
- H2 with preloaded datasets [h2.zip](#)
- [Git](#)

Clone this repository

Before start, please clone this repository in order to download all the files

```
git clone https://github.com/ontop/ontop-tutorial.git
cd ontop-tutorial
```

Program

1. [Basics of VKG Modeling](#)
 - [Mapping the first data source](#)
 - [Mapping the second data source](#)
2. [Deploying an Ontop SPARQL endpoint](#)
 - [Using Ontop CLI](#)
 - [Using Ontop Docker image](#)
3. [Interacting with an Ontop SPARQL endpoint](#)
 - [Command Line Tools \(curl, http\)](#)
 - [Python and Jupyter Notebook](#)

4. Mapping Engineering
 - Role of primary keys
 - Role of foreign keys
 - Choice of the URI templates
 - Bonus: existential reasoning
5. Materialization using Ontop
 - How to materialize data into a graph database using Ontop
6. Federating multiple databases
 - Ontop with Dremio
 - Ontop with Denodo
7. Lenses
 - Basic Lens
 - Join Lens
 - SQL Lens
 - Union Lens
 - Flatten Lens
8. External tutorials

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[Database and Ontop Setup →](#)

Community

People

- Active Core Team Members
- Core Team Emeriti
- Contributors
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Pull request
- Documentation

People

Ontop has been initiated by [the eponymous research group at the Free University of Bozen-Bolzano](#) which is still highly involved into the development of the project and into the research around it. Over the years, new members from diverse horizons have joined the project, forming a vibrant international community.

Here we list the people who has been involved in the development of the Ontop project. For the research side, [see the dedicated page](#).

Active Core Team Members

The development of Ontop is guided by an international team (reverse lexicographical order).

- [Guohui Xiao](#) (Southeast University, Ontopic)
- [Albulen Pano](#) (Unibz)
- [Davide Lanti](#) (Unibz)
- [Roman Kontchakov](#) (Birkbeck)
- [Sarah Komla-Ebri](#) (Ontopic)
- [Julien Corman](#) (Unibz)
- [Benjamin Cogrel](#) (Ontopic)
- [Diego Calvanese](#) (Unibz, Ontopic)

Core Team Emeriti

Here we honor some no-longer-active core team members who have made important contributions in the past.

- [Mariano Rodriguez-Muro](#) (Google, USA)
- [Martin Rezk](#) (Google, USA)
- [Josef Hardi](#) (Stanford University, USA)
- [Elem Güzel Kalaycı](#) (Virtual Vehicle Research GmbH, Austria)

- [Elena Botoeva](#) (University of Kent, UK)

Contributors

Some members of the Ontop community have so enriched it, that they deserve special mention.

- Lukas Sundqvist
- Mindaugas Slusnys
- Damian Rovara
- Sergey Pugacs
- Tomas Kovachev
- Dag Hovland
- Manfred Gerstgrasser
- Uğur Dönmez
- Timea Bagosi

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[Organizations](#) →

Research

Research team

[Researchers](#)
[Projects](#)
[Publications](#)

Theory

[Notation](#)
[Formal characterization of IQs](#)
[Intermediate query optimization](#)

Research team

Ontop is a project initiated by Diego Calvanese and Mariano Rodríguez-Muro in 2009 in [the KRDB research centre for Knowledge and Data at the Free University of Bolzano](#) .

A decade after, Ontop remains very active, both from a research and a software development point of view.

Researchers

Here we list the people who have been involved in the research side of the Ontop project. For the development side, see [the community page](#).

At the Free University of Bozen-Bolzano (Unibz)

- [Prof. Dr. Diego Calvanese](#) (Scientific Coordinator)
- Assistant Prof. Dr. Julien Corman
- Assistant Prof. Dr. Davide Lanti
- Assistant Prof. Dr. Alessandro Mosca (also affiliated at CNR-ISTC [Laboratory for Applied Ontology \(LOA\)](#))

Alumni researchers from Unibz

- [Prof. Dr. Guohui Xiao](#) (Southeast University / Team Leader: Dec. 2015 -)
- Dr. Mariano Rodríguez-Muro (Google USA / Founder / Former Team Leader: 2011 – Oct. 2013)
- Dr. Martin Rezk (Google USA / Former Team Leader: Oct. 2013 – Nov. 2015)
- [Dr. Elena Botoeva](#) (University of Kent, UK)
- [Dr. Benjamin Cogrel](#) ([Ontopic](#) , Bolzano)
- [Dr. Elem Güzel Kalaycı](#) (Virtual Vehicle Research GmbH,

Austria)

- [Dr. Linfang Ding](#) (Norwegian University of Science and Technology, Norway)

International collaborations

- [Dr. Roman Kontchakov](#) (Birkbeck, University of London, UK)
- [Prof. Dr. Michael Zakharyashev](#) (Birkbeck, University of London, UK)
- [Dr. Vladislav Ryzhikov](#) (Birkbeck, University of London, UK)
- [Dr. Dag Hovland](#) (Institutt for informatikk Universitetet i Oslo, Norway)

Projects

The development and the research around Ontop over the years have been mainly funded by research projects.

Current projects

- [CyclOps](#) (Automated end-to-end data life cycle management for FAIR data integration processing and re-use), Horizon Europe Project
- [D2G2](#) (Dense and Deep Geographic Virtual Knowledge Graphs for Visual Analysis), funded by DFG and the province of Bolzano
- [CRIMA](#) (Evidence-based Management of Climate Risks), FESR Project

Past projects

- Thinking ONtologiES (TONES), EU FET project
- [Optique](#) (Scalable End-User Access to Big Data) FP7 project
- Ontology-based Data Access to NoSQL Database (OBDAM), Unibz project
- Ontology-based analysis of temporal and streaming data (OBATS), Unibz project
- IDEE (data integration in the energy sector), FESR Project

- GeoVKG (geo-spatial VKG), Unibz project
- High Quality Data Integration with Ontologies (QUADRO), Unibz project
- High quality Open data Publishing and Enrichment (HOPE), Italian national PRIN (Research Projects of National Relevance)
- PACMEL (VKG for process mining), CHIST-ERA EU Project
- INODE (data management infrastructures), H2020 EU project

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[Publications](#) →

Development

- Overview
- How-tos
- Internals
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Overview

The development chapter is targeted to those interested in the development of the Ontop framework.

How-tos

- Conventions for Git
- Build Ontop
- Make a new release
- Extend Ontop to support a new database system
- Debug:
 - Protégé
 - Jetty with RDF4J workbench

Internals

Information about the internals of Ontop

- Intermediate Query data structure
- Guice Conventions

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Git workflow →

Work with us

if you are enthusiastic in the Ontop system and the Virtual Knowledge Graph technology, there are many opportunities, which range from a small course project, to a complete PhD study, or a full time developer position.

Free University of Bozen-Bolzano

At the [In2Data research group](#) at the KBDB group of Free University of Bozen-Bolzano, we are offering:

- Student course project
- Bachelor/Master thesis
- PhD student positions (<https://www.inf.unibz.it/krdp/phd/>)
- Research assistant / Scientific programmer / PostDoc researcher positions

If you are interested, please get in touch with [Prof. Diego Calvanese](#) .

Ontopic

At the company [Ontopic s.r.l.](#) , we are offering:

- Student internship
- Bachelor/Master thesis
- Part time or full time developer positions.

If you are interested, please get in touch with inquiry@ontopic.ai .

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Databricks

Supported since 5.0.0.

Through the [Databricks](#) SparkSQL connector, Ontop is able to construct VKGs on a Databricks cloud database. All main Ontop features are supported for Databricks.

Limitations & Exceptions

WARNING

Databricks does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Databricks using each supported driver:

- `com.databricks.client.jdbc.Driver`

```
jdbc.url =  
jdbc:databricks://dummyid.cloud.databricks.com:443;Htt  
  
jdbc.user = token  
jdbc.password = private-access-token  
jdbc.driver=com.databricks.client.jdbc.Driver
```

- `com.simba.spark.jdbc.Driver` (legacy driver)

```
sh
jdbc.url =
jdbc:spark://dummyid.cloud.databricks.com:443/default;

jdbc.user = token
jdbc.password = private-access-token
jdbc.driver=com.simba.spark.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the SparkSQL array type `Array<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the JSON-encoded string columns. For these columns, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In SparkSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
GET_JSON_OBJECT(TO_JSON(my_struct), '$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← AWS Redshift

DB2 →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- H2
- MariaDB
- MS SQL Server

Google BigQuery

Supported since 5.0.2.

Through the [BigQuery](#) connector, Ontop is able to construct VKGs on cloud-based Google BigQuery databases using the [Simba BigQuery JDBC Driver](#) .

Limitations & Exceptions

WARNING

Integrity constraint information is a recent feature in BigQuery. Beware that the majority of tables in BigQuery are still not providing this kind of information. Make sure to provide missing integrity constraint information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

The Simba BigQuery JDBC Driver officially only supports Java 8 and 11. If you want to use Java 17, the following option must be added to the JVM `--add-opens java.base/java.nio=ALL-UNNAMED` .

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.
- The position counter in the [flatten lens](#) starts counting at 0 instead of 1 for BigQuery.
- Accessing struct fields with the "dot operator" is not supported (see below).
- In BigQuery, columns of the type `ARRAY<ARRAY<T>>` are not supported. Use `ARRAY<STRUCT<ARRAY<T>>>` instead.

- The Ontop option `ontop.exposeSystemTables` is not supported for BigQuery, as the JDBC does not allow us to access system tables.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to BigQuery:

```
sh
jdbc.url =
jdbc:bigquery://dummydomain.com:443;ProjectId=dummypro

jdbc.property.OAuthServiceAcctEmail =
service.account@dummydomain.com
jdbc.property.OAuthPvtKey =
paht/to/private/key/file.json
jdbc.driver = com.simba.googlebigquery.jdbc.Driver
```

NOTE

The Google BigQuery JDBC supports different types of authentication methods. The above `.property` file corresponds to the *OAuthType 0: Service Account* method, which involves creating a service account for BigQuery and downloading its private key from the web interface. However, any other authentication method can be used with Ontop as well.

Nested Type Support

Ontop implements explicit compatibility with the `ARRAY<T>` type. When used with the `flatten lens`, it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON columns in the database. For such use cases, the column first has to be converted to an array type.

NOTE

The position counter starts counting at 0 instead of 1 for BigQuery.

WARNING

In BigQuery, columns cannot directly be "arrays of arrays". To represent a two-dimensional array, the type

`ARRAY<STRUCT<ARRAY<T>>>` has to be used instead of just `ARRAY<ARRAY<T>>`

Struct Access

In BigQuery, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute    JSON_VALUE(TO_JSON(my_struct),
'$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← DuckDB](#)

[H2 →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS Redshift

Supported since 5.0.2.

Through the **Redshift** connector, Ontop is able to construct VKGs on AWS Redshift databases.

Limitations & Exceptions

WARNING

AWS Redshift does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- The Simba Redshift JDBC does not support the use of default databases when connecting to Redshift.
- Accessing **SUPER** object fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample **.properties** file that can be used to connect Ontop to Redshift:

```
jdbc.url =
jdbc:redshift:iam://default.dummyaccount.eu-central-1.redshift-serverless.amazonaws.com:5439/defaultdatabase
jdbc.property.AccessKeyID = public-access-key
jdbc.property.SecretAccessKey = private-access-key
jdbc.driver = com.amazon.redshift.jdbc42.Driver
```

NOTE

The AWS Redshift JDBC supports different types of authentication methods. The above `.property` file corresponds to authentication with access keys. However, any other authentication method can be used with Ontop as well. In these cases, the corresponding connection properties have to be supplied as `jdbc.property.<property-name>` instead of the `AccessKeyID` and `SecretAccessKey` properties.

Nested Type Support

Ontop implements explicit compatibility with the Redshift type `SUPER`.

This type can be used with the [flatten lens](#). However, Ontop is not able to automatically infer the type of the result column.

Struct Access

In Redshift, individual `SUPER` struct objects can be accessed by SQL expressions using the "dot operator" on the `SUPER` column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
JSON_EXTRACT_PATH_TEXT(JSON_SERIALIZE(my_struct),
'my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

DuckDB

Supported since 5.0.2.

Through the [DuckDB](#) connector, Ontop is able to construct VKGs on DuckDB database files.

Limitations & Exceptions

- DuckDB database files prior to version 1.0 are not guaranteed to be compatible with later versions.
- Setting a default schema is not supported when connecting to DuckDB.
- Nested data types are only supported starting from version 0.7 of DuckDB.
- Accessing struct fields using the "dot operator" is not supported.
- The Ontop option `ontop.exposeSystemTables` is not supported for DuckDB, as the JDBC does not allow us to access system tables.
- DuckDB has a very specific [concurrency model](#) . In particular, it does not support concurrent multi-process read-write queries.
- DuckDB supports integrity constraints, but they tend to [costly for large tables](#) . Consider instead specifying constraints using lenses for large datasets.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DuckDB:

```
jdbc.url = jdbc:duckdb:path/to/database.db
jdbc.driver = org.duckdb.DuckDBDriver
```

sh

Nested Type Support

Ontop implements explicit compatibility with the DuckDB array type `T[]`. When used with the [flatten lens](#), it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

NOTE

Nested data types are only supported starting from version 0.7 of DuckDB.

Struct Access

Individual struct objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

WARNING

Accessing struct fields using the "dot operator" is not supported.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Dremio](#)
[Google BigQuery →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS DynamoDB

Supported (beta) since 5.1.0.

Through the [DynamoDB](#) connector, Ontop is able to construct VKGs on AWS DynamoDB tables using the [CData JDBC Driver for Amazon DynamoDB](#) .

Limitations & Exceptions

WARNING

DynamoDB only provides limited information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

To run Ontop with the CData DynamoDB JDBC, "enhanced SQL queries" must be enabled. This is achieved by adding `supportenhancedsql=true` to the connection properties.

- Nested data structures and the *FlattenLens* are not supported.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DynamoDB:

```
jdbc.url =
jdbc:amazondynamodb:AuthScheme=AwsRootKeys;AWSRegion=F
```

```
jdbc.property.AWSAccessKey = public-access-key  
jdbc.property.AWSSecretKey = private-access-key  
jdbc.driver =  
cdata.jdbc.amazondynamodb.AmazonDynamoDBDriver
```

Nested Type Support

Nested data types and the [flatten lens](#) are not supported for DynamoDB.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← AWS Athena](#)

[AWS Redshift →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▼

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

TDengine

Supported since 5.4.0.

Through the TDengine connector, Ontop is able to construct VKGs on TDengine databases.

Limitations & Exceptions

- As a time-series database, TDengine expects that all tables have a timestamp as their first column which is also used as a primary key.
- TDengine has limited support regarding date/time types. In particular, it does not support the `TIMESTAMP WITH TIME ZONE`, `DATE`, and `TIME` types and instead only uses `TIMESTAMP`
- Some data types, mainly `DECIMAL` and the `UNSIGNED` variants of integer types, are only supported through the WebSocket connection and are not currently supported by Ontop.

Database Connection

There are three ways to connect Ontop to a TDengine database using the JDBC driver:

- Native connection: depends on the client driver (libtaos.so on Linux, taos.dll on Windows, and libtaos.dylib on macOS). Below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the native driver:

```
jdbc.url = jdbc:TAOS://hostname:6030/dbname
jdbc.driver = com.taosdata.jdbc.TSDBDriver
```

sh

- WebSocket connection: does not require any additional client driver. Below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the WebSocket connection:

```
jdbc.url = jdbc:TAOS-WS://hostname:6041/dbname
jdbc.driver =
com.taosdata.jdbc.ws.WebSocketDriver
```

sh

- Restful connection: it is now recommended to use the WebSocket connection instead, as it achieves better performance. Nevertheless, it is still supported so below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the Restful connection:

```
jdbc.url = jdbc:TAOS-RS://hostname:6041/dbname
jdbc.driver = com.taosdata.jdbc.rs.RestfulDriver
```

sh

NOTE

While all three connection methods are supported, the WebSocket connection is the recommended one, as it provides the best performance without any client compatibility issues.

Nested Type Support

TDengine has some limited support for JSON data through the `JSON` data type. The main limitation is that the JSON data type can only be used in tag columns of supertables, and it cannot have a nested structure (i.e., arrays or objects inside the JSON are not allowed).

Ontop does not support nested data types or `flatten lenses` for TDengine.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← SparkSQL](#)

[Trino →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Denodo

Through the **Denodo** connector, Ontop is able to construct VKGs on Denodo databases.

Limitations & Exceptions

- SPARQL *UUID* and *hash* functions are not supported for Denodo.
- Nested data types and the **flatten lens** are not supported for Denodo.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Denodo:

```
# For Denodo >= 8
jdbc.url = jdbc:denodo://localhost:9999/
# For Denodo < 8
#jdbc.url = jdbc:vdb://localhost:9999/books
jdbc.user = admin
jdbc.password = ${denodo.password}
jdbc.driver = com.denodo.vdp.jdbc.Driver
```

Nested Type Support

Nested data types and the **flatten lens** are not supported for Denodo.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← DB2

Dremio →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Dremio

Supported since 4.1.0.

Through the [Dremio](#) connector, Ontop is able to construct VKGs on Dremio databases. Ontop can connect to normal Dremio instances, as well as Dremio Cloud databases.

Limitations & Exceptions

WARNING

Dremio does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

The Dremio JDBC officially only supports Java 8 and 11. If you want to use Java 17, the following option must be added to the JVM `--add-opens java.base/java.nio=ALL-UNNAMED`.

- The `position` argument for the [flatten lens](#) cannot be used with Dremio.
- When using a Dremio version below 21.0.0, some larger queries that employ `UNION`s may fail. This can be solved by adding `ontop.maxNbChildrenLiftingDBFS=0` to the properties

Database Connection

The following shows the content of a sample `file`

`.properties`

that can be used to connect Ontop to Dremio for each supported JDBC:

- `com.dremio.jdbc.Driver`

```
jdbc.url = jdbc:dremio:direct=localhost:31010
jdbc.user = user
jdbc.password = password
jdbc.driver = com.dremio.jdbc.Driver
```

sh

- `org.apache.arrow.driver.jdbc.ArrowFlightJdbcDriver`

```
jdbc.url = jdbc:arrow-flight-sql://localhost:32010
jdbc.user = dremio
jdbc.password = password
jdbc.driver =
org.apache.arrow.driver.jdbc.ArrowFlightJdbcDriver
```

sh

Nested Type Support

Ontop implements explicit compatibility with the Dremio array type `LIST` which can be used with the `flatten` lens. However, it cannot infer the type of the flattened result column.

To use a `flatten` lens on a JSON-encoded array stored in the database, it first has to be converted to a `LIST`.

WARNING

The `position` argument for `flatten` lenses cannot be used with Dremio.

Struct Access

Individual struct objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Denodo](#)

[DuckDB →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

SparkSQL

Supported since 4.2.0.

Through the [SparkSQL](#) connector, Ontop is able to construct VKGs on an external Spark database, using the SparkSQL JDBC.

Limitations & Exceptions

WARNING

SparkSQL does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to SparkSQL using the JDBC Hive Driver:

```
jdbc.url=jdbc:hive2://localhost:10000
jdbc.driver = org.apache.hive.jdbc.HiveDriver
jdbc.user = user
jdbc.password = password
```

sh

Nested Type Support

Ontop implements explicit compatibility with the SparkSQL array

type `Array<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the JSON-encoded string columns. For these columns, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In SparkSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
```

```
GET_JSON_OBJECT(TO_JSON(my_struct), '$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Snowflake](#)

[TDengine →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Trino

Supported since 5.0.2.

Through the [Trino](#) connector, Ontop is able to construct VKGs on Trino databases.

Limitations & Exceptions

WARNING

Trino does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Trino:

```
jdbc.url =
jdbc:trino://localhost:8080/defaultdatabase
jdbc.user = user
jdbc.password = password
jdbc.driver = io.trino.jdbc.TrinoDriver
```

In case Trino is set up without authentication, the `jdbc.user` and `jdbc.password` lines can also be left out.

Nested Type Support

Ontop implements explicit compatibility with the Trino array type `array(t)`. When used with the [flatten lens](#), it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

Individual `MAP` structures can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

`ROW` structures can be accessed using the `[position]` operator. `position` must be an integer index starting from 1:

```
SELECT my_struct[1] FROM ...
```

WARNING

Accessing object fields using the "dot operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← TDengine](#)

[FAQ →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Presto

Supported since 5.0.2.

Through the [Presto](#) connector, Ontop is able to construct VKGs on Presto databases.

Limitations & Exceptions

WARNING

Presto does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Presto:

```
jdbc.url =
jdbc:presto://localhost:8080/defaultdatabase
jdbc.user = user
jdbc.password = password
jdbc.driver =
com.facebook.presto.jdbc.PrestoDriver
```

In case Presto is set up without authentication, the `jdbc.user` and `jdbc.password` lines can also be left out.

Nested Type Support

Ontop implements explicit compatibility with the Presto array type `array(t)`. When used with the `flatten lens`, it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

In Presto, individual `ROW` objects can be accessed by SQL expressions using the "dot operator" on the `ROW` column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_row.my_attribute | JSON_EXTRACT(CAST(my_struct AS
JSON), '$.my_attribute')
```

Elements of `MAP` objects can be accessed using the `[]` operator, as in

```
SELECT my_map['my_attribute'] FROM ...
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← PostgreSQL](#)

[Snowflake →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS Athena

Supported since 5.0.2.

Through the [Athena](#) connector, Ontop is able to construct VKGs on AWS Athena databases using the [Simba Athena JDBC Driver](#) .

Limitations & Exceptions

WARNING

Athena does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- The Simba Athena JDBC does not support the use of default databases when connecting to Athena.
- Accessing object fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Athena:

```
jdbc.url = jdbc:awsathena://AwsRegion=eu-central-1;S3OutputLocation=s3://result-location
jdbc.user = user
jdbc.password = password
jdbc.driver = com.simba.athena.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the Athena array type `ARRAY<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

Individual `MAP` objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

`ROW` objects, can also be accessed using the `[key]` operator. Please note, that in this case, `key` must be an integer index instead of the name of the field:

```
SELECT my_struct[1] FROM ...
```

WARNING

Accessing object fields using the "dot operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Generic JDBC \(not recommended\)](#)

[AWS DynamoDB →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys

Lenses

- Example
- Document root
- Lens
- Constraints

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Lenses

Since 4.2.0 (was experimental in 4.1.x)

Prior to 5.0.0, lenses were named Ontop views.

Lenses are relational views defined at the level of Ontop and unknown to the underlying database. Lenses can be defined from database relations and from other lenses.

As database relations, lenses have a name which can be used in the source part of the mapping entries. They are specified in a separate file that can be provided to Ontop through a dedicated parameter (`--lenses` for the [CLI commands](#) that support it, `ONTOP_LENSES_FILE` for the [Docker image](#)).

At the moment, 5 types of lenses are available:

1. [Basic lenses](#) (defined over one base relation)
2. [Join lenses](#) (defined over multiple base relations)
3. [SQL lenses](#) (defined from an arbitrary SQL query)
4. [Union lenses](#) (defined as a union of multiple base relations)
5. [Flatten lenses](#) (defined as an unnest operation over one base relation)

Don't use lenses in complex source SQL queries

The Ontop mapping SQL parser only parses simple forms of SQL queries (without unions, aggregations, limits, order by, etc.). Non-parsed queries are treated as black-box views, that is as strings that are injected into the final SQL queries sent to the database. If some lenses appear in these black-box views, the resulting SQL queries will be rejected by the database because they refer to relations it does not know.

One interesting feature of lenses is that you can specify additional

[constraints](#) holding on them (in addition to the ones that can be inferred from base relations). The constraints can be:

- [Unique constraints](#)
- [Other functional dependencies](#)
- [Foreign keys](#)
- [Non-null constraints](#) (columns that do not include null values)
- [IRI-safe constraints](#) (columns on which the R2RML safe encoding has no effect). *Since 5.0.2.*

Tutorial

To see lenses used in practice, check [the tutorial](#)

Example

```

{
  "relations": [
    {
      "name":
["\"lenses\"", "\"hr\"", "\"persons\""],
      "baseRelation":
["\"hr\"", "\"persons\""],
      "filterExpression": "\"firstName\" IS
NOT NULL AND \"lastName\" IS NOT NULL",
      "columns": {
        "added": [
          {
            "name": "\"fullName\"",
            "expression":
"CONCAT(UPPER(\"firstName\"), ' ', \"lastName\")"
          }
        ],
        "hidden": [
          "\"firstName\"",
          "\"lastName\""
        ]
      },
      "uniqueConstraints": {
        "added": [
          {
            "name": "uc2",

```

```

        "determinants":
["\"ssn\""]
    }
]
},
"otherFunctionalDependencies": {
    "added": [
        {
            "determinants":
["\"regionOfResidence\""],
            "dependents":
["\"countryOfResidence\""]
        }
    ],
    "foreignKeys": {
        "added": [
            {
                "name": "fk1",
                "from":
["\"regionOfResidence\""],
                "to": {
                    "relation":
["\"geo\"", "\"regions\""],
                    "columns":
["\"reg_id\""]
                }
            }
        ],
        "nonNullConstraints": {
            "added": [
                "\"email\""
            ],
        },
        "iriSafeConstraints": {
            "added": [
                "\"ssn\""
            ],
        },
        "type": "BasicLens"
    },
    {
        "name": [
            "\"lenses\"",
            "\"rooms\""
        ],
    },

```

```
      "join": {
        "relations": [
          ["rooms"],
          ["lenses", "hotels"]
        ],
        "columnPrefixes": [
          "r_",
          "h_"
        ]
      },
      "filterExpression":
        "\"r_hotel_id\"=\"h_id\" AND (\"h_stars\" = '***'
        OR \"h_price\" = '€€€') AND \"r_guests\" = 2",
      "columns": {
        "added": [
        ],
        "hidden": [
        ]
      },
      "type": "JoinLens"
    },
    {
      "name":
        ["lenses", "geo", "top_region"],
      "query": "SELECT \"regionOfResidence\"
        AS \"region\", COUNT(*) FROM \"hr\".\"persons\"
        GROUP BY \"regionOfResidence\" ORDER BY COUNT(*)
        DESC LIMIT 1",
      "type": "SQLLens"
    }
  ]
}
```

Document root

The lenses document has the following JSON structure:

```
{
  "relations": [Lens]
}
```

Key	Type

relations

Array of

Lens

-s

Lens

Common fields

All the lenses accept the following fields (most of them are optional):

```
{
  "name": [String],
  "uniqueConstraints": {
    "added": [UniqueConstraint]
  },
  "otherFunctionalDependencies": {
    "added": [OtherFunctionalDependency]
  },
  "foreignKeys": {
    "added": [ForeignKey]
  },
  "nonNullConstraints": {
    "added": [String]
  },
  "iriSafeConstraints": {
    "added": [String]
  },
  "type": String
}
```

json

Key	Type
name	Array of Strings
uniqueConstraints	JSON Object
uniqueConstraints.added	Array of UniqueConstr
otherFunctionalDependencies	JSON Object
otherFunctionalDependencies.added	Array of OtherFunctionalDeper

	s
foreignKeys	JSON Object
foreignKeys.added	Array of ForeignKey
nonNullConstraints	JSON Object
nonNullConstraints.added	Array of Strings
iriSafeConstraints	JSON Object
iriSafeConstraints.added	Array of Strings
type	String

BasicLens

A basic lens is defined from one base (parent) relation, over which it can apply a filter, an extended projection and additional constraints.

In addition to the common fields, basic lenses accept the following ones:

```
{
  "baseRelation": [String],
  "columns": {
    "added": [AddedColumn],
    "hidden": [String]
  },
  "filterExpression": String,
```

```
"type": "BasicLens"
}
```

Key	Type	Description
<code>baseRelation</code>	Array of Strings	Name components of the base relation (with correct quoting)
<code>columns</code>	JSON Object	Optional (since 5.0.2)
<code>columns.added</code>	Array of <code>AddedColumns</code> s	
<code>columns.hidden</code>	Array of Strings	Names of the columns from the base relation to be projected away (with correct quoting)
<code>filterExpression</code>	String	Expression expressed in the SQL dialect of the data source. Can only refer to columns from the base relation, not to added columns. Can be empty. Optional

`AddedColumn`

Added columns have the following definition:

```
{
  "name": String,
}
```

json

```

    "expression": String
  }

```

Key	Type	Description
name	String	New column name (with correct quoting)
expression	String	SQL expression defining the column. Can only refer to columns from the base relations, not to added columns

JoinLens

A join lens is defined from multiple base relations, over which it can apply a filter (joining condition), an extended projection and additional constraints.

A prefix is assigned to each base relation and is added as a prefix to their column names. This allows to avoid conflicts due to columns with the same names in base relations.

In addition to the common fields, join lenses accept the following ones:

```

{
  "join": {
    "relations": [[String]],
    "columnPrefixes": [String]
  },
  "columns": {
    "added": [AddedColumn],
    "hidden": [String]
  },
  "filterExpression": String,
  "type": "JoinLens"
}

```

Key	Type	Descripti
join	JSON Object	

<code>join.relations</code>	Array of arrays of Strings	Arrays of the components each base re (with correct quoting)
<code>join.columnPrefixes</code>	Array of Strings	Prefix for each base relation applied on its column name. Follows the same order as <code>join.relations</code>
<code>columns</code>	JSON Object	Optional (since 5.0.2)
<code>columns.added</code>	Array of <code>AddedColumns</code>	
<code>columns.hidden</code>	Array of Strings	Names of the columns from base relation to be projected (with correct quoting)
<code>filterExpression</code>	String	Expression expressed in SQL dialect of data source. It can only refer to prefixed columns from the base relations, not added columns. Can be empty. Optional

SQLLens

A SQL lens is defined from an arbitrary SQL query. While expressive, it also comes with important restrictions. When applicable, other types of lenses should be used instead.

Avoid referring to lenses in the SQL query

As Ontop uses the same parser as from the mapping

source queries, the same restriction apply: non-parsed queries will be treated internally as black-box views and will fail. Please consider using other types of lenses if possible.

No unique constraint and foreign key inferred from the base relations

Please consider using other types of lenses if possible.

In addition to the [common fields](#), SQL lenses accept the following ones:

```
{
  "query": String,
  "type": "SQLLens"
}
```

json

Key	Type	Description
query	String	SQL query

UnionLens

Since 5.1.0

A union lens is defined from multiple base relations that share attributes with exactly the same names and types. The relations will be merged with each other, concatenating their contents.

When defining a union lens, a *"provenance column"* can be determined to hold, for each data entry, the name of the base relation it originates from.

In addition to the [common fields](#), union lenses accept the following ones:

```
{
  "unionRelations": [[String]],
}
```

json

```
"makeDistinct": boolean,  
"provenanceColumn": String,  
"type": "UnionLens"  
}
```

Key	Type	Description
unionRelations	Array of arrays of Strings	Arrays of the name components of each base relation (with correct quoting).
makeDistinct	boolean	Determines, if the final resulting union should be made distinct.
provenanceColumn	String	The name of the column that should contain the base relation each entry originates from. If not provided, provenance information will not be included in the result.

FlattenLens

Since 5.1.0

A flatten lens is defined from one base (parent) relation that contains an array-like data structure in one of its fields. The array is flattened into multiple rows, where each row contains a single item from the flattened array in the `new` column. Columns of the base relation not included in the `kept` list will be discarded when flattening the array.

In addition, a `position` column can be included in the lens, providing a unique index for each flattened row in its parent relation.

NOTE

The flatten operation is only performed on the "outer-most" array layer. Multi-dimensional arrays will have their dimensionality reduced by 1.

In addition to the [common fields](#), flatten lenses accept the following ones:

json

```
{
  "flattenedColumn": {
    "name": String,
    "datatype": String
  },
  "columns": {
    "kept": [String],
    "new": String,
    "position": String
  },
  "type": "FlattenLens"
}
```

Key	Type	Description
<code>flattenedColumn</code>	JSON Object	Identifies the column that is to be flattened.
<code>flattenedColumn.name</code>	String	The name of the column that is to be flattened.
<code>flattenedColumn.datatype</code>	String	The type of the column that is to be flattened.
<code>columns</code>	JSON Object	Defines the columns of the output relation.
<code>columns.kept</code>	Array of Strings	The names of the columns from the base relation that should be included in the output.
		The name of the newly

<code>columns.new</code>	String	created column that should hold the elements of the flattened array.
<code>columns.position</code>	String	The name of the newly created column that should hold the index of each flattened element in its source list. If not provided, no position column will be included.

Due to various limitations in the language definitions, the FlattenLens is currently not equally supported for all dialects. The table below lists, in detail, the level of support for each dialect. "Flatten" defines if the flatten lens is supported by the dialect, "position" defines if the `position` column can be provided, and "Infer base type" indicates if Ontop is able to infer the type of the flattened output column if the input is an array type. "Array Type" and "JSON Type" indicate if the flatten lens is supported over array-like types (`ARRAY`, `ARRAY<T>`, `LIST`, `T[]` etc.) and JSON-arrays (either as `JSON` type or as `VARCHAR`) respectively.

Dialect	Flatten	Position	Infer base type	Array Type
AWS Athena	✓	✓	✓	✓
AWS DynamoDB	✗	✗	✗	✗
AWS Redshift	✓	✓	✗	✓
BigQuery	✓	✓	✓	✓

DB2	✗	✗	✗	✓
Databricks	✓	✓	✓	✓
Denodo	✗	✗	✗	✓
Dremio	✓	✗	✗	✓
DuckDB	✓	✓	✓	✓
MariaDB	✓	✓	✗	✗
MS SQLServer	✓	✗	✗	✗
MySQL	✓	✓	✗	✗
Oracle	✓	✓	✗	✗
PostgreSQL	✓	✓	✓	✓
Presto	✓	✓	✓	✓
Snowflake	✓	✓	✗	✓
SparkSQL	✓	✓	✓	✓
TDengine	✗	✗	✗	✗
Trino	✓	✓	✓	✓

Constraints

UniqueConstraint

```
{
  "name": String,
  "determinants": [String]
}
```

json

Key	Type	Description
<code>name</code>	String	Name of the unique

		constraint
determinants	Array of Strings	Column names (with correct quoting)

OtherFunctionalDependency

Useful for dealing with denormalized data, where unique constraints cannot be applied.

```
{
  "determinants": [String],
  "dependents": [String]
}
```

json

Key	Type	Description
determinants	Array of Strings	Column names (with correct quoting) that determine the values of dependent columns
dependents	Array of Strings	Column names (with correct quoting) whose values are determined by determinant columns

ForeignKey

```
{
  "name": String,
  "from": [String],
  "to": {
    "relation": [String],
    "columns": [String]
  }
}
```

json

Key	Type	Description
name	String	Name of the foreign key
	Array	Source columns (with correct

<code>from</code>	of Strings	quoting)
<code>to</code>	JSON Object	
<code>to.relation</code>	Array of Strings	Name components of the target relation (with correct quoting)
<code>to.columns</code>	Array of Strings	Target columns (with correct quoting). Same order as for the source columns

NotNullConstraint

```
{  
  "added": [String]  
}
```

json

Key	Type	Description
<code>added</code>	Array of Strings	List of names of non-nullable columns. One String per column

IRISafeConstraint

```
{  
  "added": [String]  
}
```

json

Key	Type	Description
<code>added</code>	Array of Strings	List of names of IRI-safe columns. One String per column

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Research

- Research team
- Publications
 - Journals
 - Conferences
 - Workshops
 - Tutorials and Demos
 - Master theses

Theory

- Notation
- Formal characterization of IQs
- Intermediate query optimization

Publications

Journals

1. Virtual Knowledge Graphs: An Overview of Systems and Use Cases. Guohui Xiao, Linfang Ding, Benjamin Cogrel, Diego Calvanese. Data Intelligence. 2019.
2. Ontop-spatial: Ontop of Geospatial Databases. Konstantina Bereta, Guohui Xiao, and Manolis Koubarakis. Journal of Web Semantics. 2019.
3. Ontology-Based Access To Temporal Data With Ontop: A Framework Proposal. Elem Güzel Kalaycı, Sebastian Brandt, Diego Calvanese, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyashev. International Journal of Applied Mathematics and Computer Science. 2019.
4. VIG: Data Scaling for OBDA Benchmarks. Davide Lanti, Guohui Xiao, and Diego Calvanese. Semantic Web Journal. 2019.
5. Ontology-based Data Access - Beyond relational sources. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, Guohui Xiao. Intelligenza Artificiale. 2019.
6. Querying Log Data with Metric Temporal Logic. Sebastian Brandt, Elem Güzel Kalaycı, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyashev. Journal of Artificial Intelligence Research, 2018.
7. Ontop: Answering SPARQL Queries over Relational Databases. Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Semantic Web Journal. 2017 (*SWJ 2016 Outstanding Paper Award*)
8. Ontology Based Data Access in Statoil. Evgeny Kharlamov, Dag Hovland, Martin G. Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis E. Ioannidis, Yannis Kotidis, Manolis Koubarakis, Arild Waaler.

- Journal of Web Semantics, 2017.
9. Efficient SPARQL-to-SQL with R2RML mappings. Mariano Rodriguez-Muro and Martin Rezk, J. of Web Semantics. 2015
10. Optique --- zooming in on big data access. Martin Giese, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Özgür Özçep, Martin Rezk, Riccardo Rosati, Ahmet Soylu, Guillermo Vega-Gorgojo, Arild Waaler, and Guohui Xiao. IEEE Computer. 2015

Conferences

1. The Virtual Knowledge Graph System Ontop. Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalayci, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. ISWC (Resource), 2020.
2. Semantic Integration of Bosch Manufacturing Data Using Virtual Knowledge Graphs. Elem Güzel Kalaycı, Irlan Grangel González, Felix Lösch, Guohui Xiao, Anees ul-Mehdi, Evgeny Kharlamov, and Diego Calvanese. ISWC (In-Use), 2020.
3. Enriching Ontology-based Data Access with Provenance. Diego Calvanese, Davide Lanti, Ana Ozaki, Rafael Penaloza, and Guohui Xiao. IJCAI, 2019.
4. Efficient Handling of SPARQL OPTIONAL for OBDA. Guohui Xiao, Roman Kontchakov, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. ISWC (Research), 2018.
5. Ontology-based data access: A survey. Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, Michael Zakharyashev. IJCAI. 2018.
6. Ontop-temporal: A Tool for Ontology-based Query Answering over Temporal Data. Elem Güzel Kalayci, Guohui Xiao, Vladislav Ryzhikov, Tahir Emre Kalayci, and Diego Calvanese. CIKM. 2018.
7. Efficient Ontology-Based Data Integration with canonical IRIs. Guohui Xiao, Dag Hovland, Dimitris Bilidas, Martin Rezk, Martin Giese, and Diego Calvanese. ESWC, 2018.
8. Cost-Driven Ontology-Based Data Access. Davide Lanti, Guohui Xiao, Diego Calvanese. ISWC (Research), 2017.
9. Semantic Rule-Based Equipment Diagnostics. Gulnar Mehdi,

- Evgeny Kharlamov, Ognjen Savkovic, Guohui Xiao, Elem Güzel Kalayci, Sebastian Brandt, Ian Horrocks, Mikhail Roshchin, Thomas A. Runkler. ISWC (In-Use), 2017. ([Best In-Use Paper Award](#))
10. Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic. Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, Michael Zakharyashev. AAAI, 2017.
11. Beyond OWL 2 QL in OBDA: Rewritings and Approximations. Elena Botoeva, Diego Calvanese, Valerio Santarelli, Domenico Fabio Savo, Alessandro Solimando and Guohui Xiao. AAAI, 2016.
12. Ontology-Based Data Access for Maritime Security. Stefan Brüggemann, Konstantina Bereta, Guohui Xiao and Manolis Koubarakis. ESWC (In-Use), 2016.
13. OBDA Constraints for Effective Query Answering. Dag Hovland, Davide Lanti, Martin Rezk and Guohui Xiao. RuleML, 2016.
14. Towards Practical OBDA with Temporal Ontologies (Position Paper). Diego Calvanese, Elem Güzel Kalayci, Vladislav Ryzhikov and Guohui Xiao. RR, 2016.
15. Ontology-based Integration of Cross-linked Datasets. Diego Calvanese, Martin Giese, Dag Hovland and Martin Rezk (ISWC 2015), 2015.
16. The NPD benchmark: Reality check for OBDA systems. Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. In Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT 2015). ACM Press, 2015.
17. Answering SPARQL Queries over Databases and under OWL 2 QL Entailment Regime. Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyashev. International Semantic Web Conference (ISWC), 2014.
18. Rules and Ontology Based Data Access. Guohui Xiao, Martin Rezk, Mariano Rodriguez-Muro, and Diego Calvanese. Eighth International Conference on Web Reasoning and Rule Systems, (RR), 2014.
19. High Performance Query Answering over DL-Lite Ontologies.

- Mariano Rodriguez-Muro, Diego Calvanese. KR 2012
20. Quest: Efficient SPARQL-to-SQL for RDF and OWL. Mariano Rodriguez-Muro, Josef Hardi, Diego Calvanese. ISWC (Posters & Demos) 2012

Workshops

1. Fast and Simple Data Scaling for OBDA Benchmarks. Davide Lanti, Guohui Xiao and Diego Calvanese. BLINK@ISWC, 2016.
2. OBDA Beyond Relational DBs: A Study for MongoDB. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk and Guohui Xiao. Description Logics, 2016.
3. The ontop framework for ontology based data access. Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, Mindaugas Slusnys, and Guohui Xiao. CSWS (Posters and Demos), Wuhan, China, August 2014.
4. The NPD benchmark for OBDA systems. Davide Lanti, Martin Rezk, Mindaugas Slusnys, Guohui Xiao, and Diego Calvanese. SSWS 2014
5. A scalable benchmark for OBDA systems: Preliminary report. Diego Calvanese, Davide Lanti, Martin Rezk, Mindaugas Slusnys, and Guohui Xiao. ORE 2014
6. Query Rewriting and Optimisation with Database Dependencies in Ontop. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev: Description Logics 2013
7. Evaluating SPARQL-to-SQL Translation in Ontop. Mariano Rodriguez-Muro, Martín Rezk, Josef Hardi, Mindaugas Slusnys, Timea Bagosi, Diego Calvanese: ORE 2013
8. OBDA with Ontop. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev: ORE 2013
9. Ontop at Work. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev. OWLED 2013
10. On Rewriting, Answering Queries in OBDA Systems for Big Data (Short Paper). Diego Calvanese, Ian Horrocks, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Michael Meier, Mariano Rodriguez-Muro, Dmitriy Zheleznyakov. OWLED 2013
11. Quest, an OWL 2 QL Reasoner for Ontology-based Data

Access. Mariano Rodriguez-Muro, Diego Calvanese. OWLED 2012

Tutorials and Demos

1. [Destination Tutorial](#) given at the Knowledge Graph Conference 2021 by Diego Calvanese, Benjamin Cogrel and Guohui Xiao.
2. [Novel Developments in Ontology-Based Data Access and Integration \(NOBDI\)](#) Diego Calvanese, Benjamin Cogrel and Guohui Xiao. Full day tutorial, 17th International Conference of the Italian Association for Artificial Intelligence -- Trento, November 20-23, 2018.
3. [Semantic Technologies for Data Access and Integration](#). Diego Calvanese and Guohui Xiao. Full day tutorial, CIKM. Torino, 26 October 2018.
4. [Ontology-based Data Access: Theory and Practice](#) . Guohui Xiao and Roman Konchakov. Full day tutorial, IJCAI 2018 Tutorial. Stockholm, July 14, 2018.
5. [Mapping Management and Expressive Ontologies in Ontology-Based Data Access](#) . Diego Calvanese, Benjamin Cogrel and Guohui Xiao. Full day tutorial, EKAW. Bologna, November 19, 2016.
6. [Ontology-Based Data Access with Ontop](#) . Benjamin Cogrel. Lecture at the ESSLLI summer school, 2016.
7. [Ontop: Answering SPARQL Queries over Relational Databases](#) . Guohui Xiao. Stanford, 2016.
8. [Ontology-based Data Access: From Theory to Practice](#) (Diego Calvanese, Martin Rezk, Guohui Xiao. ISWC 2015 Tutorial)
9. [Ontology-based data access --- Theory and Practice](#) . (Guohui Xiao, Summer School at CSWS2014, Wuhan, China)
10. [An Introduction to Description Logics and Query Rewriting](#). Roman Kontchakov, Michael Zakharyashev. Reasoning Web 2014
11. [Ontology-Based Data Access with Databases: A Short Course](#). (Roman Kontchakov, Mariano Rodriguez-Muro, Michael Zakharyashev. Reasoning Web 2013)
12. [Ontop: A Tutorial](#) (Mariano Rodriguez-Muro, Vienna'13)

Master theses

- Lukas Sundqvist: [Extending VKG Systems with RDF-star Support](#) , Free University of Bozen-Bolzano, March 2022.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Research team](#)

[Notation →](#)

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Using lenses

In this part of the tutorial, we cover the use of lenses with Ontop. Lenses allow us to create "virtual views" of specific operations over the data, that can be referenced in our mapping definitions for a more structured access to the database.

We will go over all lenses currently supported by Ontop:

1. [Basic Lens](#)
2. [Join Lens](#)
3. [Union Lens](#)
4. [Flatten Lens](#)
5. [SQL Lens](#)

NOTE

In this tutorial, we use a different database than in the previous sections. Before running any of the examples, please follow the [setup instructions](#).

Not supported by Ontop-Protégé

Lenses are not supported by the Ontop plugin for Protégé. To follow this tutorial, please use a text editor.

General

Lenses are provided to Ontop as a JSON file of the following structure:

```
{  
  "relations": [Lens]  
}
```

json

where each `Lens` object corresponds to one particular lens, holding a *name* through which it can be referenced in the mapping and a *type* which indicates the specific type of the lens. We can include any number of lenses in this file, and they can reference database relations as well as each other.

Some more common features of lenses will be discussed in the [Basic Lens](#) section.

In each of these sections, we will construct one *mapping* file and one *lenses* file that can be passed to Ontop. To simplify testing, a common template for all mapping files is already included in the tutorial files. You can continue the preparation of the individual mappings from this template.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Ontop with Teiid](#)

[Database and Ontop Setup →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database
 - Solutions
 - Choosing the Right Approach for Your Use Case

How to materialize data into a graph database

In this section, we present two ways to materialize your Knowledge Graph using Ontop.

Solutions

1. Materialize in RDF files and load into a triplestore

For the first solution, you will need the following prerequisites:

- Access to the tutorial H2 database
- The `mapping file` `university-complete.obda` and the `properties file` `university-complete.properties`
- `Ontop CLI`

Using the CLI command `ontop-materialize` (<https://ontop-vkg.org/guide/cli#ontop-materialize>), you can `materialize` your KG into one or multiple files. For simplicity, we keep the default option and only materialize it into one file.

```
sh
./ontop materialize
--mapping=university-complete.obda \
--properties=university-complete.properties \
-f turtle \
-o materialized-triples.ttl
```

After running the command, we have all the content of our KG copied to the file `materialized-triples.ttl`.

Now we load this file in the triplestore of our choice, in this case, we use `GraphDB`. This graph database offers `several ways to load files`. Here, since our file is small, we go for the simplest option

and load it directly from the UI.

Once this is done, we can query this KG using GraphDB.

2. Deploy a VKG and fetch its content from the graph database

For the second solution, we make use of the concept of KG virtualization.

We deploy the KG as a virtual KG first and then query it from the graph database. In this way, you can retrieve the triples and store them locally in the graph database.

Triples are directly streamed to the graph database: no intermediate file storage is involved, making this solution more direct than the previous one.

let's deploy the KG as a virtual KG using the *ontop-endpoint* command:

```
sh
./ontop endpoint
--mapping=input/university-complete.obda \
--properties=input/university-complete.properties \
```

Now Ontop is deployed as a [SPARQL endpoint](http://localhost:8080/sparql) available at <http://localhost:8080/sparql>.

Let's go now to GraphDB. To fetch and insert all the triples from the VKG exposed by Ontop, we run the following SPARQL INSERT query from GraphDB itself:

```
INSERT {
  ?s ?p ?o
}
WHERE {
  SERVICE <http://localhost:8080/sparql> {
    ?s ?p ?o
  }
}
```

This query materializes the same triples as with the first approach.

Choosing the Right Approach for Your Use Case

1. Small Dataset, Easy Communication: If your dataset isn't large and you can easily set up communication between the Ontop SPARQL endpoint and the graph database, go with solution #2. It avoids dealing with files and intermediate storage.
2. Large Dataset, Efficient Loading: For very large datasets, choose the most efficient loading solution supported by the triplestore, even if it requires more effort to set up.
3. Materializing Fragments of the KG: Solution #2 allows easy materialization of specific fragments of the Knowledge Graph by adapting the SPARQL query. You can have hybrid KGs with some parts stored in the graph database and the rest kept virtual.
4. Advantage of Keeping Data Virtual: Keeping data virtual is great for large volumes of sensor data that constantly update. It's better to keep this part virtual while storing rich contextual information in the graph database.

Ontology Usage

If you're familiar with Ontop, you might have noticed that we didn't use an ontology in this example. Providing an ontology to Ontop can result in a significantly larger KG due to the reasoning capabilities embedded in Ontop. However, GraphDB also has reasoning capabilities, allowing reasoning to be done later in GraphDB, making materialization simpler and faster. If your graph database doesn't support reasoning, Ontop can handle it.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Configuration keys

Here is a list of the configuration keys which can be set in the properties file.

Key	Type	Description

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← [HTTP caching](#) [Lenses](#) →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Ontop views

Ontop views have been renamed lenses. See [the new page](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)

Query logging

- Example
- Entries
- Default settings
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Query logging

Since 4.1.0.

Query logging, disabled by default, prints one-line JSON objects into the standard output.

Its JSON format follows some Elasticsearch conventions and works nicely with the corresponding stack (FileBeat, ES, Kibana).

Example

json

```
{
  "@timestamp": "2023-10-13T14:02:05.973Z",
  "message": "query:all",
  "application": "ontop-odh",
  "payload": {
    "queryId": "36ade6c6-f091-4af2-9828-ee6316e191f4",
    "classesUsedInQuery": [
      "http://schema.org/Hotel"
    ],
    "propertiesUsedInQuery": [
      "http://schema.org/name"
    ],
    "tables": [
      "\"v_accommodationsopen\""
    ],
    "reformulationDuration": 50,
    "reformulationCacheHit": false,
    "user": "joe",
    "groups": "admins,users",
    "roles": "my_app_write,my_app_reader",
    "httpHeaders": {
      "referer": "http://localhost:8080/"
    },
    "extractedQueryTemplate": {
      "hash":
"de167012e1ce3cf9dce66a986a3343d883a1f319f800132565b61"
```



```
      "parameters": {
        "v0": "\"Residence Tuberis\""
      },
      "sparqlQuery": "PREFIX schema:
<http://schema.org/>\nSELECT * WHERE {\n  ?h a
schema:Hotel ; schema:name ?name .\n  FILTER
(langMatches(lang(?name), 'en'))\n  FILTER(str(?
name) = \"Residence Tuberis\")\n} \nLIMIT 10",
      "reformulatedQuery": "ans1(h,name)\nCONSTRUCT
[h, name] [name/\"Residence Tuberis\"^^@en,
h/RDF(http://noi.example.org/data/accommodation/{
(VARCHARToTEXT(Id1m148)),IRI)]\n  NATIVE
[Id1m148]\nSELECT v1.\"Id\" AS \"Id1m148\"\nFROM
\"v_accommodationsopen\" v1\nWHERE ('Residence
Tuberis' = v1.\"AccoDetail-en-Name\" AND
'HotelPension' = v1.\"AccoTypeId\")\nLIMIT 10\n",
      "executionBeforeUnblockingDuration": 8,
      "executionAndFetchingDuration": 30,
      "totalDuration": 80,
      "resultCount": 10
    }
  }
}
```

Entries

Key	Type	
@timestamp	Timestamp	
message	String	Type of the message. Its value is the query. If an exception occurs, its value is the exception message.
application	String	Controlled by the application. Its value is the application name.
payload	JSON object	Gathers the payload of the query.
queryId	UUID	Unique to each query.
classesUsedInQuery	Array of IRIs	IRIs of the classes used in the query.

<code>propertiesUsedInQuery</code>	Array of IRIs	IRIs of the pre query.
<code>tables</code>	Array of Strings	Names of the
<code>reformulationDuration</code>	Integer	Query reform
<code>reformulationCacheHit</code>	Boolean	True if the ref
<code>user</code>	String	Since 5.2.0. U
<code>groups</code>	String	Since 5.2.0. U Disabled by c
<code>roles</code>	String	Since 5.2.0. U Disabled by c
<code>httpHeaders</code>	JSON object	Values of a sp this list is emp the property <code>ontop.query</code> to true.
<code>extractedQueryTemplate. hash</code>	String	Hash of the q removing ext similar querie
<code>extractedQueryTemplate. parameters</code>	JSON object	Constants ext
<code>sparqlQuery</code>	String	SPARQL quer
<code>reformulatedQuery</code>	String	Includes the S
<code>executionBefore UnblockingDuration</code>	Integer	Duration betw query is sent
<code>executionAndFetching Duration</code>	Integer	Duration betw query is sent
<code>totalDuration</code>	Integer	Total durator
<code>resultCount</code>	Integer	Number of re

Default settings

```
# Query logging is disabled by default
ontop.queryLogging=false
```

```
# Application name (needed for logging)
ontop.applicationName=Ontop

# Includes the SPARQL query string into the query
log
ontop.queryLogging.includeSparqlQuery=true
# Includes the reformulated query into the query
log
ontop.queryLogging.includeReformulatedQuery=false
# Includes classes and properties into the query
log
ontop.queryLogging.includeClassesAndProperties=true

# Includes DB tables/views into the query log
ontop.queryLogging.includeTables=true
# Includes the user ID, his/her groups and roles
ontop.queryLogging.includeUserInfo=false
# Provides separated message at different phases
(after reformulation, result set unblocked, last
result fetched)
ontop.queryLogging.decomposition=false
# Sets that merged messages are only inserted when
decomposition is disabled
ontop.queryLogging.decompositionAndMergingMutuallyExcl
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Predefined query endpoint \(beta\)](#)

[HTTP caching →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Remarks
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

HTTP caching

Since 4.1.0.

HTTP caching allows to reduce the load on the SPARQL endpoint and to provide faster responses to clients. It is particularly useful when many clients issue the same queries and data is not updated at a high frequency.

To enable HTTP caching, add the property `ontop.http.cacheControl` in the properties file. This will cause the insertion of a `Cache-Control` HTTP header in successful responses, with the specified value.

Example

```
ontop.http.cacheControl=max-age=60, stale-while-revalidate=20, stale-if-error=300
```

This entry informs the cache that values can be considered as fresh for 60 seconds. After this period, the cache remains allowed to serve cached results for 20 more seconds while revalidating the requests in the background, or for 300 more seconds in case of error with the upstream server.

Ontop does not embed a HTTP cache. We recommend setting up a HTTP cache like [Varnish](#) or [Nginx](#) above the Ontop endpoint.

Remarks

- Most caches (e.g. Nginx but also Web browsers) do not cache POST requests by default. Since most SPARQL queries are sent

with the POST method, it is important to turn this option on.
This option is safe as Ontop is a read-only system.

- Consider using the GET method for sending SPARQL queries so as to take advantage from Web browser caches.
- For an example of configuration for Nginx, see [this file](#) .

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Query logging](#)

[Configuration keys →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Build Ontop Bundles

Requirements

- a Linux, Mac OS X or Windows machine;
- git, for checking out Ontop code (refer to installation instructions for your OS/distribution);
- Java 11 (JDK, not JRE, refer to installation instructions for your OS/distribution);
- JAVA_HOME environment variable correctly set up and pointing to a Java 11 JDK. Run `echo $JAVA_HOME` (on Windows: `echo %JAVA_HOME%`) and check the path printed (if any). If the variable is unset, you can set it temporarily in the terminal session using `export JAVA_HOME=[path]` (on Windows: `setx JAVA_HOME -m "[path]"`), where [path] points to the JDK installation (the directory containing bin, lib, etc). This setting may be made persistent by editing your `~/.bash_profile` (on Windows, edit System and Security > System > Advanced System Settings > Environment Variables). In particular, for Mac OS X

```
$ vim ~/.bash_profile
Add this line: export
JAVA_HOME=$(/usr/libexec/java_home)
$ source ~/.bash_profile
```

- Maven 3.6.0 or later, with `mvn` command available on `PATH`. Run

```
$ mvn -v
Apache Maven 3.9.0
(9b58d2bad23a66be161c4664ef21ce219c2c8584)
Maven home: /opt/maven/current
```

```
Java version: 11.0.17, vendor: Debian, runtime:
/usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "6.1.0-3-amd64",
arch: "amd64", family: "unix"
```

Check the versions of Maven (3.9.0 > 3.6.0 in the example) and Java (11.0.17, i.e. Java 11).

Checkout Ontop source code

In a terminal, move to the parent directory where you want to checkout Ontop, and run

```
$ git clone https://github.com/ontop/ontop.git
$ cd ontop
```

Build all bundles

Starting from Ontop 4.1, from the root directory of Ontop source code tree, run

```
$ mvn -Dmaven.test.skip -Prelease
```

The command compiles Ontop modules and generates the Ontop bundles distributed on GitHub, whose files are written in two directories.

Under `build/distribution/target/`:

- `ontop-cli-[ontop-version].zip` - libraries/scripts to use ontop from the command line and setup a SPARQL endpoint;

Under `protege/distribution/target/`:

- `it.unibz.inf.ontop.protege-[ontop-version].jar` - plugin for Protégé 5.x installation files (drop it in Protégé's plugins folder);
- `ontop-protege-bundle-platform-independent-[ontop-
version].zip`

- `[version].zip` - Protégé 5.x bundled with Ontop plugin and ready to run (just unzip and use the `run.sh` or `run.bat` scripts);
- `ontop-protege-bundle-linux-[ontop-version].zip` - as above, for Linux;
- `ontop-protege-bundle-os-x-[ontop-version].zip` - as above, for Mac OS X;
- `ontop-protege-bundle-win-[ontop-version].zip` - as above, for Windows.

Notes:

- profile activation `-Prelease` enables the generation of all the Ontop bundles as well as source code and Javadoc JARs needed for publishing on Maven Central (if this profile is omitted, only JAR files for Ontop binaries are created);
- option `-Dmaven.test.skip` disables the compiling and execution of unit and integration tests (generally performed at release time) to speed up the process - omit it to enable tests, e.g., for testing changes to source code (note: may also use `-DskipTests` that disables executing tests, but still compiles them);
- the build may be performed also by running script `build-release.sh` (`build-release.cmd` on Windows), which works also in older Ontop versions (4.0.3 or earlier). In these older versions, the script generates bundles in sub-directories `ontop-cli`, `ontop-protege`, and `ontop-webapps` of directory `build/distribution`, and also generates bundles for Jetty 9 + RDF4J workbench + Ontop and for Tomcat 8 + RDF4J workbench + Ontop. These bundles have been deprecated in Ontop 4 and completely removed since Ontop 5 .

Build specific bundles

Starting from version `4.1` and further revised in version `5.0.2`, we provide Maven profiles to build specific Ontop bundles as quickly as possible (with respect to using the above 'build all' command), which may come in handy when modifying

Ontop source code. Specifically:

- to build the Ontop command line tool only, i.e., `ontop-cli-[ontop-version].zip` (note: up to version `5.0.1`, `asset-cli` was `cli` and tests were disabled by default)

```
$ mvn -Dmaven.test.skip -Passet-cli
```

- to build the Ontop Protégé plugin, i.e.,

```
it.unibz.inf.ontop.protege-[ontop-version].jar
```

(profile available since version `5.0.2`)

```
$ mvn -Dmaven.test.skip -Passet-plugin
```

- to build the Ontop Protégé plugin

```
it.unibz.inf.ontop.protege-[ontop-version].jar
```

as well as the `ontop-protege-bundle-[platform]-[ontop-version].zip` Protégé bundles including it (note: up to version `5.0.1`, `asset-protege` was `protege` and tests were disabled by default)

```
$ mvn -Dmaven.test.skip -Passet-protege
```

The Maven profiles `asset-cli`, `asset-plugin`, `asset-protege` employed in the commands above exclude unneeded artifacts from the build, which is thus faster. These profiles can be arbitrarily combined to produce multiple bundles, e.g., `mvn -Passet-cli,asset-protege` to generate both command line tool and Protégé bundles (from version 4.1 up to version 5 excluded, it was possible to build the RDF4 workbench + Ontop files by enabling profile `webapps`, now removed).

Compiling the Ontop Protégé plugin

Starting from version `5.0.2`, it is necessary to enable profile `plugin` (or a profile depending on it, such as `asset-plugin`, `asset-protege`, or `release`) in order to compile `ontop-`

`protege-plugin`. Note that commands `mvn compile` and `mvn test` will not work with profile `plugin` (or a depending profile) since compiling/testing `ontop-protege-plugin` requires building `ontop-protege-dependencies:shaded` first to [relocate Guava via the Maven shade plugin](#), but this happens only during the `package` lifecycle phase of Maven, i.e., starting from `mvn package` or later phase. Unfortunately, relocating Guava is needed to avoid conflicts with the older Guava version used in Protégé and cannot be avoided, and doing that before the `package` phase is not feasible. Therefore, developers should always use `mvn package -Pplugin` or later phase when working with `ontop-protege-plugin`, even when just compiling or testing is intended.

Other useful Maven commands

The following Maven commands can be used to facilitate some build-related development tasks, the involved plugins (e.g., `license-maven-plugin`) being already configured in Ontop root `pom.xml` file:

- `mvn license:add-third-party` - produces a report file `build/distribution/THIRD-PARTY.txt` useful to verify the use of Java libraries with compatible licenses
- `mvn org.codehaus.mojo:versions-maven-plugin:display-plugin-updates` - check for Maven plugins that can be updated
- `mvn org.codehaus.mojo:versions-maven-plugin:display-dependency-updates` - check for dependencies that can be updated
- `mvn dependency:analyze-duplicate` - check there are no duplicate dependencies declared in `pom.xml` files
- `mvn dependency:analyze-dep-mgt` - check that dependency versions under centralized `<dependencyManagement>` sections are not overridden in child `pom.xml` files
- `mvn dependency:analyze` - check for *used and undeclared* dependencies and for *unused and declared* dependencies (both situations that should be avoided)
- `mvn com.github.ekryd.sortpom:sortpom-maven-`

`plugin:sort` - sorts section of `pom.xml` files, to improve consistency

- `mvn net.revelc.code.formatter:formatter-maven-plugin:format` - auto-formats the `pom.xml` files, to improve consistency

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Git workflow](#)

[Make a new release →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Generic JDBC (not recommended)

WARNING

Don't rely on generic JDBC support. It isn't sufficient for any data source.

Ontop provides a generic implementation, that is used as default for any JDBC. While part of the functionality may work right out of the box with this default implementation, there is no guarantee that Ontop will be able to handle any database setup and SPARQL queries correctly, especially once special limitations, non standard datatypes and exceptions come into play.

However, Ontop also provides dialect-specific implementations for a variety of different database management systems and SQL dialects. The following documents describe the process of connecting Ontop to each of them, as well as special points and caveats for specific cases.

In this section, we provide general information and important points for each of the supported systems.

NOTE

If your SQL dialect of interest is not included in this list, it is possible to implement a connector for it, following [this guide](#).

NOTE

To use Ontop with any database system, its corresponding JDBC driver is required in the `jdbc` directory, which is passed to the Ontop instance as

described in the [setup guide](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Lenses](#)

[AWS Athena →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS Athena

Supported since 5.0.2.

Through the [Athena](#) connector, Ontop is able to construct VKGs on AWS Athena databases using the [Simba Athena JDBC Driver](#) .

Limitations & Exceptions

WARNING

Athena does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- The Simba Athena JDBC does not support the use of default databases when connecting to Athena.
- Accessing object fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Athena:

```
jdbc.url = jdbc:awsathena://AwsRegion=eu-central-1;S3OutputLocation=s3://result-location
jdbc.user = user
jdbc.password = password
jdbc.driver = com.simba.athena.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the Athena array type `ARRAY<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

Individual `MAP` objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

`ROW` objects, can also be accessed using the `[key]` operator. Please note, that in this case, `key` must be an integer index instead of the name of the field:

```
SELECT my_struct[1] FROM ...
```

WARNING

Accessing object fields using the "dot operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Generic JDBC \(not recommended\)](#)

[AWS DynamoDB →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS DynamoDB

Supported (beta) since 5.1.0.

Through the [DynamoDB](#) connector, Ontop is able to construct VKGs on AWS DynamoDB tables using the [CData JDBC Driver for Amazon DynamoDB](#) .

Limitations & Exceptions

WARNING

DynamoDB only provides limited information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

To run Ontop with the CData DynamoDB JDBC, "enhanced SQL queries" must be enabled. This is achieved by adding `supportenhancedsql=true` to the connection properties.

- Nested data structures and the *FlattenLens* are not supported.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DynamoDB:

```
jdbc.url =
jdbc:amazondynamodb:AuthScheme=AwsRootKeys;AWSRegion=F
```



```
jdbc.property.AWSAccessKey = public-access-key  
jdbc.property.AWSSecretKey = private-access-key  
jdbc.driver =  
cdata.jdbc.amazondynamodb.AmazonDynamoDBDriver
```

Nested Type Support

Nested data types and the [flatten lens](#) are not supported for DynamoDB.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← AWS Athena](#)

[AWS Redshift →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

AWS Redshift

Supported since 5.0.2.

Through the **Redshift** connector, Ontop is able to construct VKGs on AWS Redshift databases.

Limitations & Exceptions

WARNING

AWS Redshift does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- The Simba Redshift JDBC does not support the use of default databases when connecting to Redshift.
- Accessing **SUPER** object fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample **.properties** file that can be used to connect Ontop to Redshift:

```
jdbc.url =  
jdbc:redshift:iam://default.dummyaccount.eu-  
central-1.redshift-  
serverless.amazonaws.com:5439/defaultdatabase  
jdbc.property.AccessKeyID = public-access-key  
jdbc.property.SecretAccessKey = private-access-key  
jdbc.driver = com.amazon.redshift.jdbc42.Driver
```

sh

NOTE

The AWS Redshift JDBC supports different types of authentication methods. The above `.property` file corresponds to authentication with access keys. However, any other authentication method can be used with Ontop as well. In these cases, the corresponding connection properties have to be supplied as `jdbc.property.<property-name>` instead of the `AccessKeyID` and `SecretAccessKey` properties.

Nested Type Support

Ontop implements explicit compatibility with the Redshift type `SUPER`.

This type can be used with the [flatten lens](#). However, Ontop is not able to automatically infer the type of the result column.

Struct Access

In Redshift, individual `SUPER` struct objects can be accessed by SQL expressions using the "dot operator" on the `SUPER` column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
JSON_EXTRACT_PATH_TEXT(JSON_SERIALIZE(my_struct),
'my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Databricks

Supported since 5.0.0.

Through the [Databricks](#) SparkSQL connector, Ontop is able to construct VKGs on a Databricks cloud database. All main Ontop features are supported for Databricks.

Limitations & Exceptions

WARNING

Databricks does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Databricks using each supported driver:

- `com.databricks.client.jdbc.Driver`

```
jdbc.url =  
jdbc:databricks://dummyid.cloud.databricks.com:443;Htt  
  
jdbc.user = token  
jdbc.password = private-access-token  
jdbc.driver=com.databricks.client.jdbc.Driver
```

- `com.simba.spark.jdbc.Driver` (legacy driver)

```
sh
jdbc.url =
jdbc:spark://dummyid.cloud.databricks.com:443/default;

jdbc.user = token
jdbc.password = private-access-token
jdbc.driver=com.simba.spark.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the SparkSQL array type `Array<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the JSON-encoded string columns. For these columns, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In SparkSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
GET_JSON_OBJECT(TO_JSON(my_struct), '$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← AWS Redshift

DB2 →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

DB2

Through the IBM DB2 connector, Ontop is able to construct VKGs on DB2 databases.

Limitations & Exceptions

- SPARQL *UUID* and *hash* functions are not supported for DB2.
- Nested data types and the *flatten lens* are not supported for DB2

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DB2:

```
jdbc.url =  
jdbc:db2://localhost:50000/defaultdatabase  
jdbc.user = user  
jdbc.password = password  
jdbc.driver = com.ibm.db2.jcc.DB2Driver
```

Nested Type Support

Nested data types and the *flatten lens* are not supported for DB2.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Denodo

Through the **Denodo** connector, Ontop is able to construct VKGs on Denodo databases.

Limitations & Exceptions

- SPARQL *UUID* and *hash* functions are not supported for Denodo.
- Nested data types and the **flatten lens** are not supported for Denodo.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Denodo:

```
# For Denodo >= 8
jdbc.url = jdbc:denodo://localhost:9999/
# For Denodo < 8
#jdbc.url = jdbc:vdb://localhost:9999/books
jdbc.user = admin
jdbc.password = ${denodo.password}
jdbc.driver = com.denodo.vdp.jdbc.Driver
```

Nested Type Support

Nested data types and the **flatten lens** are not supported for Denodo.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← DB2

Dremio →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

Dremio

Supported since 4.1.0.

Through the [Dremio](#) connector, Ontop is able to construct VKGs on Dremio databases. Ontop can connect to normal Dremio instances, as well as Dremio Cloud databases.

Limitations & Exceptions

WARNING

Dremio does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

The Dremio JDBC officially only supports Java 8 and 11. If you want to use Java 17, the following option must be added to the JVM `--add-opens`
`java.base/java.nio=ALL-UNNAMED`.

- The `position` argument for the [flatten lens](#) cannot be used with Dremio.
- When using a Dremio version below 21.0.0, some larger queries that employ `UNION`s may fail. This can be solved by adding `ontop.maxNbChildrenLiftingDBFS=0` to the properties

Database Connection

The following shows the content of a sample file

```
.properties
```

that can be used to connect Ontop to Dremio for each supported JDBC:

- `com.dremio.jdbc.Driver`

```
jdbc.url = jdbc:dremio:direct=localhost:31010
jdbc.user = user
jdbc.password = password
jdbc.driver = com.dremio.jdbc.Driver
```

sh

- `org.apache.arrow.driver.jdbc.ArrowFlightJdbcDriver`

```
jdbc.url = jdbc:arrow-flight-sql://localhost:32010
jdbc.user = dremio
jdbc.password = password
jdbc.driver =
org.apache.arrow.driver.jdbc.ArrowFlightJdbcDriver
```

sh

Nested Type Support

Ontop implements explicit compatibility with the Dremio array type `LIST` which can be used with the `flatten` lens. However, it cannot infer the type of the flattened result column.

To use a flatten lens on a JSON-encoded array stored in the database, it first has to be converted to a `LIST`.

WARNING

The `position` argument for flatten lenses cannot be used with Dremio.

Struct Access

Individual struct objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Denodo](#)

[DuckDB →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

DuckDB

Supported since 5.0.2.

Through the DuckDB connector, Ontop is able to construct VKGs on DuckDB database files.

Limitations & Exceptions

- DuckDB database files prior to version 1.0 are not guaranteed to be compatible with later versions.
- Setting a default schema is not supported when connecting to DuckDB.
- Nested data types are only supported starting from version 0.7 of DuckDB.
- Accessing struct fields using the "dot operator" is not supported.
- The Ontop option `ontop.exposeSystemTables` is not supported for DuckDB, as the JDBC does not allow us to access system tables.
- DuckDB has a very specific **concurrency model** . In particular, it does not support concurrent multi-process read-write queries.
- DuckDB supports integrity constraints, but they tend to **costly for large tables** . Consider instead specifying constraints using lenses for large datasets.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to DuckDB:

```
jdbc.url = jdbc:duckdb:path/to/database.db
jdbc.driver = org.duckdb.DuckDBDriver
```

sh

Nested Type Support

Ontop implements explicit compatibility with the DuckDB array type `T[]`. When used with the [flatten lens](#), it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

NOTE

Nested data types are only supported starting from version 0.7 of DuckDB.

Struct Access

Individual struct objects can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

WARNING

Accessing struct fields using the "dot operator" is not supported.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Dremio](#)
[Google BigQuery →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- H2
- MariaDB
- MS SQL Server

Google BigQuery

Supported since 5.0.2.

Through the [BigQuery](#) connector, Ontop is able to construct VKGs on cloud-based Google BigQuery databases using the [Simba BigQuery JDBC Driver](#) .

Limitations & Exceptions

WARNING

Integrity constraint information is a recent feature in BigQuery. Beware that the majority of tables in BigQuery are still not providing this kind of information. Make sure to provide missing integrity constraint information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

The Simba BigQuery JDBC Driver officially only supports Java 8 and 11. If you want to use Java 17, the following option must be added to the JVM `--add-opens java.base/java.nio=ALL-UNNAMED` .

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.
- The position counter in the [flatten lens](#) starts counting at 0 instead of 1 for BigQuery.
- Accessing struct fields with the "dot operator" is not supported (see below).
- In BigQuery, columns of the type `ARRAY<ARRAY<T>>` are not supported. Use `ARRAY<STRUCT<ARRAY<T>>>` instead.

- The Ontop option `ontop.exposeSystemTables` is not supported for BigQuery, as the JDBC does not allow us to access system tables.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to BigQuery:

```
sh
jdbc.url =
jdbc:bigquery://dummydomain.com:443;ProjectId=dummypro

jdbc.property.OAuthServiceAcctEmail =
service.account@dummydomain.com
jdbc.property.OAuthPvtKey =
paht/to/private/key/file.json
jdbc.driver = com.simba.googlebigquery.jdbc.Driver
```

NOTE

The Google BigQuery JDBC supports different types of authentication methods. The above `.property` file corresponds to the *OAuthType 0: Service Account* method, which involves creating a service account for BigQuery and downloading its private key from the web interface. However, any other authentication method can be used with Ontop as well.

Nested Type Support

Ontop implements explicit compatibility with the `ARRAY<T>` type. When used with the `flatten lens`, it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON columns in the database. For such use cases, the column first has to be converted to an array type.

NOTE

The position counter starts counting at 0 instead of 1 for BigQuery.

WARNING

In BigQuery, columns cannot directly be "arrays of arrays". To represent a two-dimensional array, the type

`ARRAY<STRUCT<ARRAY<T>>>` has to be used instead of just `ARRAY<ARRAY<T>>`

Struct Access

In BigQuery, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute    JSON_VALUE(TO_JSON(my_struct),
'$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← DuckDB](#)

[H2 →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- MariaDB
- MS SQL Server

H2

Through the [H2](#) connector, Ontop is able to construct VKGs on H2 databases, in memory, and on files.

Limitations & Exceptions

- SPARQL *hash* functions are not supported for H2.
- Nested data types and the [flatten lens](#) are not supported for H2

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to H2:

```
jdbc.url = jdbc:h2:mem:database-name
jdbc.user = user
jdbc.password = password
jdbc.driver = org.h2.Driver
```

Nested Type Support

Nested data types and the [flatten lens](#) are not supported for H2.

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
 - Limitations & Exceptions
 - Database Connection
 - Nested Type Support
- MS SQL Server

MariaDB

Supported since 5.0.0.

Through the [MariaDB](#) connector, Ontop is able to construct VKGs on MariaDB databases.

Limitations & Exceptions

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to MariaDB:

```
jdbc.url =  
jdbc:mariadb://localhost:3306/defaultdatabase  
jdbc.user = user  
jdbc.password = password  
jdbc.driver = org.mariadb.jdbc.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the MariaDB type `JSON` which can be used with the [flatten lens](#). However, it cannot infer the type of the flattened result column.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [MariaDB's JSON functions](#) can be used to further work with them.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← H2](#)

[MS SQL Server →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB

MS SQL Server

- Limitations & Exceptions
- Database Connection
- Nested Type Support

MS SQL Server

Through the [MS SQL Server](#) connector, Ontop is able to construct VKGs on SQL Server databases.

Limitations & Exceptions

- RegEx SPARQL functions are not supported.
- The `position` argument for the [Flatten Lens](#) cannot be used with `SqlServer`.
- Support for `DATETRUNC` in SQL Server is very recent (introduced in SQL Server 2022)

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to `SqlServer`:

```
sh
jdbc.url =
jdbc:sqlserver://localhost:1433;databaseName=defaultda

jdbc.user = user
jdbc.password = password
jdbc.driver =
com.microsoft.sqlserver.jdbc.SQLServerDriver
```

Nested Type Support

Nested data types are not supported by `SqlServer`. However, Ontop allows the usage of the [flatten lens](#) over string columns that contain JSON-encoded arrays. The output type of the flattened result column cannot be inferred.

WARNING

The `position` argument for flatten lenses cannot be used with SQLServer.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [SQLServer's JSON functions](#) can be used to further work with them.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MariaDB](#)

[MySQL →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server

MySQL

- Limitations & Exceptions
- Database Connection

MySQL

Through the [MySQL](#) connector, Ontop is able to construct VKGs on MySQL databases.

Limitations & Exceptions

- String literals have to be encased in *single quotes* (`'`). Double quote string literals (`"`) are not supported.
- Accessing JSON object fields with the "dot operator" is not supported (see below).
- MySQL 5.x comes with some limitations (e.g. no anonymous bnode support, no regexp replace, no nested data support). MySQL 5.x reached its end-of-life, so please consider upgrading.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to MySQL:

```
sh
jdbc.url =
jdbc:mysql://localhost:3306/defaultdatabase?
useCursorFetch=true
jdbc.user = user
jdbc.password = password
jdbc.driver = com.mysql.cj.jdbc.Driver
```

WARNING

The parameter `useCursorFetch=true` needs to be added to the JDBC URL for streaming to be enabled. Otherwise, the JDBC driver will block until the full result set is fetched, which may cause out-of-memory

exceptions when materializing or running large queries.

Nested Type Support

Ontop implements explicit compatibility with the MySQL type `JSON` which can be used with the [flatten lens](#). However, it cannot infer the type of the flattened result column.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [MySQL's JSON functions](#) can be used to further work with them.

WARNING

Accessing object fields using the "arrow operator" is not supported.

WARNING

Nested type only supported for MySQL ≥ 8.0

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MS SQL Server](#)

[Oracle →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- Limitations & Exceptions

Oracle

Through the [Oracle](#) connector, Ontop is able to construct VKGs on external Oracle databases.

Limitations & Exceptions

- Ontop cannot perform inference based on nested data types in Oracle.
- Ontop represents the "big integer" datatype as `NUMBER(19)`.

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Oracle:

```
jdbc.url =  
jdbc:oracle:thin:@//localhost:49161/defaultcontainer  
  
jdbc.user = user  
jdbc.password = password  
jdbc.driver = oracle.jdbc.OracleDriver
```

Nested Type Support

Ontop does not implement explicit compatibility with nested datatypes in Oracle. However, Ontop allows the usage of the [flatten lens](#) over string columns that contain JSON-encoded arrays. The output type of the flattened result column cannot be inferred.

To use the flatten lens with Oracle array types, they must first be converted to JSON strings.

In case the flattened column is still a nested structure after the flatten operation is performed (e.g. for arrays of objects), [Oracle's JSON functions](#) can be used to further work with them.

WARNING

Accessing object fields using the "dot operator" is not supported. Please use SQL JSON functions instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← MySQL](#)

[PostgreSQL →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

PostgreSQL

Through the [PostgreSQL](#) connector, Ontop is able to construct VKGs on external Postgres databases.

Limitations & Exceptions

- Accessing JSON object fields with the "arrow operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to PostgreSQL:

```
jdbc.url =
jdbc:postgresql://localhost:5432/database
jdbc.user = admin
jdbc.password = password
jdbc.driver = org.postgresql.Driver
```

Nested Type Support

Ontop implements explicit compatibility with the PostgreSQL array type `T[]`. When used with the [flatten](#) lens, it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the `JSON` and `JSONB` datatypes, which are both recognized by Ontop. For these types, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In PostgreSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
```

```
JSON_EXTRACT_PATH(TO_JSON(my_struct), '$.my_attribute')
```

WARNING

The "arrow operators" used to access JSON objects in PostgreSQL are not supported in Ontop. Please use JSON functions instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← [Oracle](#)

[Presto](#) →

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Presto

Supported since 5.0.2.

Through the [Presto](#) connector, Ontop is able to construct VKGs on Presto databases.

Limitations & Exceptions

WARNING

Presto does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Presto:

```
jdbc.url =
jdbc:presto://localhost:8080/defaultdatabase
jdbc.user = user
jdbc.password = password
jdbc.driver =
com.facebook.presto.jdbc.PrestoDriver
```

In case Presto is set up without authentication, the `jdbc.user` and `jdbc.password` lines can also be left out.

Nested Type Support

Ontop implements explicit compatibility with the Presto array type `array(t)`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

The `flatten` lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

In Presto, individual `ROW` objects can be accessed by SQL expressions using the "dot operator" on the `ROW` column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_row.my_attribute  JSON_EXTRACT(CAST(my_struct AS
JSON), '$.my_attribute')
```

Elements of `MAP` objects can be accessed using the `[]` operator, as in

```
SELECT my_map['my_attribute'] FROM ...
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← PostgreSQL](#)

[Snowflake →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

SparkSQL

Supported since 4.2.0.

Through the [SparkSQL](#) connector, Ontop is able to construct VKGs on an external Spark database, using the SparkSQL JDBC.

Limitations & Exceptions

WARNING

SparkSQL does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to SparkSQL using the JDBC Hive Driver:

```
jdbc.url=jdbc:hive2://localhost:10000
jdbc.driver = org.apache.hive.jdbc.HiveDriver
jdbc.user = user
jdbc.password = password
```

sh

Nested Type Support

Ontop implements explicit compatibility with the SparkSQL array

type `Array<T>`. When used with the `flatten` lens, it is able to automatically infer the type of the result column.

Furthermore, the flatten lens can also be used with the JSON-encoded string columns. For these columns, however, Ontop cannot infer the output type of the flattened column.

Struct Access

In SparkSQL, individual struct objects can be accessed by SQL expressions using the "dot operator" on the struct column. In Ontop, this feature is not currently supported.

Should any of their elements still be required, then a workaround can be performed by first transforming the struct into a JSON object and then accessing it using JSON functions.

Example:

```
my_struct.my_attribute
```

```
GET_JSON_OBJECT(TO_JSON(my_struct), '$.my_attribute')
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Snowflake](#)

[TDengine →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
 - Source Query
 - Target Triple Structure
 - Meta-Mapping
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Ontop Mapping Language

While Ontop is compatible with the W3C standard mapping language [R2RML](#) , it also provides its own native mapping language (OBDA).

An OBDA mapping file is a text file with the extension `.obda` and consists of two main sections:

- `PrefixDeclaration` : a list of prefix definitions used in the mapping file. Each prefix is declared by a pair of its identifier (or name) and its IRI definition.
- `MappingDeclaration` : collection of mapping assertions where each mapping assertion consists of three fields: `mappingId` , `source` and `target` . The mappingId is any string identifying the assertion, the source is an arbitrary SQL query over the database, and the target is a [triple template](#) that contains placeholders that reference column names mentioned in the source query.

The following is an example of a valid OBDA mapping file:

```
[PrefixDeclaration]
:      http://www.example.org/library#
xsd:   http://www.w3.org/2001/XMLSchema#
rdf:   http://www.w3.org/1999/02/22-rdf-syntax-ns#

[MappingDeclaration] @collection [[
mappingId      Book collection
target         :BID_{id} a :Book .
source         SELECT id FROM books

mappingId      Book title
target         :BID_{id} :title {title} .
source         SELECT id, title FROM books
]]
```

The empty lines between the two sections and between the mappings are mandatory.

Comments

To comment out a line in an OBDA mapping file, you can use the `;` character at the beginning of the line. Note that you can only comment out entire lines, not parts of them.

Source Query

The `source` query in a mapping assertion is an SQL query over the underlying relational database and as such it uses the SQL syntax of that specific database dialect. So things like quotes conventions may vary depending on the database system used. For example, in PostgreSQL double quotes are used for tables and column identifiers, while in MySQL backticks are used.

WARNING

The Ontop SQL parser only parses simple SQL queries without unions, aggregations, order by, etc. Non-parsed queries are treated as black-box views and sent directly to the database so the optimizations that Ontop can apply are limited.

Target Triple Structure

This section explains the syntax and limitations for the `target` of mapping assertions, which is an adaptation of the `Turtle` syntax. Each target triple is written like an RDF subject-predicate-object (SPO) graph.

```
target <http://www.example.org/library#BID_{id}>
rdf:type :Book .
```

[S]

```
[P]      [O]

target  <http://www.example.org/library#BID_{id}>
:title {title} .

                                [S]

[P]      [O]
```

Each triple must be separated by a space followed by a period (`s p o .`) and is composed of three nodes:

- Subject node: The subject node can be one of the following terms:
 1. IRI or blank node constant: e.g.
`<http://www.example.org/library#BID_FF125>` or `_:Library1`
 2. IRI or blank node template: e.g.
`<http://www.example.org/library#BID_{id}>` or `_{id}`
 3. IRI or blank node column: a column directly from the source query (e.g. `<{iri}>`)
- Predicate node: The predicate node can be one of the following terms:
 1. IRI constant: e.g.
`<http://www.example.org/library#title>`
 2. IRI template: e.g. `<http://www.example.org/library#{predicate}>`
 3. IRI column: a column from the source query (e.g. `<{predicate_iri}>`)

Note that the special predicate `a` is a shortcut that stands for `rdf:type` (more precisely, for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`).

- Object node: The object node can be one of the following terms:
 1. IRI or blank node constant: e.g.,
`<http://www.example.org/library#Book>`
 2. IRI or blank node template: e.g.,

- `<http://www.example.org/Author-{pid}>`
- 3. IRI or blank node column: e.g. `<{object_iri}>`
- 4. Literal constant: either an implicitly typed literal (e.g., `123` or `true` or `"John"`), an explicitly typed literal (e.g., `"John"^^xsd:string`, `"123"^^xsd:integer`) or a literal with a language tag (e.g., `"Il Trono di Spade"@it`).
- 5. Literal column: a column from the source query (e.g., `{title}`). It can also be explicitly typed (e.g., `{title}^^xsd:string`) or have a language tag (e.g., `{title}@en`).
- 6. Literal template: just like literal constants, literal templates can also be explicitly typed or have a language tag. Literal templates can be useful to create complex, arbitrary literals by concatenation (e.g., `"POINT ({longitude} {latitude})"^^geo:wktLiteral`).

IRI and blank node templates apply IRI-safe encoding to their columns, following the [R2RML standard](#). For example, if we have the IRI template

`<http://www.example.org/library#BID_{name}>` and suppose that for the `name` column we have the value `"John Library"`, then the generated IRI will be

`<http://www.example.org/library#BID_John%20Library>`.

Instead, IRI columns are not transformed, so their values are expected to already be valid IRIs.

WARNING

Literal constants, templates, and columns can either be explicitly typed or have a language tag, but the two cannot be combined. For example, the following mapping is *invalid*:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books
WHERE lang='ITALIAN'
target         :BID_{id} :title
               {title}^^xsd:string@it .
```

IRI or Blank Node Template

IRI or blank node templates are used in the target of mapping assertions for the identification of generated objects. An IRI/blank node template is a string with placeholders (e.g.

`<http://www.example.org/library#BID_{id}>`). More than one placeholder can appear in a template, which allows constructing complex paths. For example, as an IRI template:

```
mappingId      Spare parts
source         SELECT product, part, vendor FROM
product
target         <http://example.org/{vendor}/{product}/{part}> a
               :Part .
```

or as a blank node template:

```
mappingId      Spare parts
source         SELECT product, part, vendor FROM
product
target         _:{product}/{part} a :Part .
```

Prefixes in IRI or Blank Node Templates

Prefixes can be used when writing IRI or blank node templates and are replaced by their definition when Ontop parses the mappings.

Example. Assume that the following prefixes are defined:

```
: http://www.example.org/ontology1#
p: http://www.example.org/ontology2#
```

Then this mapping assertion:

```
mappingId      Example
source         SELECT col1, col2 FROM table
target         <http://www.example.org/ontology1#
```

```
{col1}> :property
<http://www.example.org/ontology2#{col2}>
```

is equivalent to this mapping assertion:

```
mappingId      Example
source         SELECT col1, col2 FROM table
target         :{col1} :title p:{col2}
```

Literal

Literal Typing

It is possible to explicitly declare the type of a literal by suffixing it with `^^` followed by the IRI of the datatype. For example:

```
mappingId      Book titles
source         SELECT id, title, edition, comment
FROM books
target         :BID_{id} :title
               {title}^^xsd:string; :edition
               {edition}^^xsd:integer; :description {comment} .
```

The type used in the mapping has to agree with the type in the ontology (if specified). If the type is not specified (for example, for the `description` property in the previous mapping), the system will look at the SQL type of the SQL column used in the mapping and will use the [Natural Mapping of SQL values](#) as defined by [R2RML standard](#) .

Language Tags

The language for a literal can be specified directly using the `@` symbol followed by the language tag. For example:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
               lang= 'ITALIAN'
target         :BID_{id} :title {title}@it .
```


WARNING

Language tags can only be constants, it is not possible to obtain them dynamically from the database. So for example, the following mapping is *invalid*:

```
mappingId      Book titles in Italian
source         SELECT id, title, lang FROM
books
target         :BID_{id} :title
               {title}@{lang} .
```

Named Graphs

By default, triples generated by a triple pattern are added to the default graph. However, it is also possible to specify a named graph by using the keyword `GRAPH` followed by an IRI constant or template and then the triple pattern in curly braces. For example:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
               lang= 'ITALIAN'
target         GRAPH
               <http://www.example.org/graphs/italian> {
               :BID_{id} :title {title}@it . }
```

or, using a template for the graph name:

```
mappingId      Book titles in Italian
source         SELECT id, title FROM books WHERE
               lang= 'ITALIAN'
target         GRAPH
               <http://www.example.org/graphs/{lang}> { :BID_{id}
               :title {title}@it . }
```

Compact Form

Following the [Turtle](#) syntax, Ontop's native mapping format allows writing an RDF graph in a compact textual form. A set of triples sharing the same subject can be written as a predicate list, where predicate-object pairs are separated by semicolons. Similarly, a set of triples sharing the same subject and predicate can be written as an object list, where objects are separated by commas.

Predicate List: These two examples are equivalent ways of writing the triple template for an Author.

```
:Author-{ID} a :Author .
:Author-{ID} :firstName {FNAME} .
:Author-{ID} :lastName {LNAME} .
:Author-{ID} :writes :Book-{ID} .
```

```
:Author-{ID} a :Author; :firstName {FNAME};
:lastName {LNAME}; :writes :Book-{ID} .
```

Object List: These two examples are equivalent ways of writing the triple template for the *A Game of Thrones* book.

```
:A_Game_of_Thrones :title "A Game of Thrones"@en-US .
:A_Game_of_Thrones :title "Il Trono di Spade"@it .
```

```
:A_Game_of_Thrones :title "A Game of Thrones"@en-US, "Il Trono di Spade"@it .
```

Meta-Mapping

Meta-mapping assertions are syntactically the same as normal assertions, but they allow users to include variables in the targets without restrictions. This means that class and property names can be constructed dynamically from the database.

Example: Consider the following mapping assertions:

```
mappingId      mapping1
target         <{iri}> a :{value}_{code} .
source         SELECT value, iri, code FROM table1
WHERE code > 0
```

```
mappingId      mapping2
target         <{iri}> :{role}_{code} {value} .
source         SELECT value, iri, code, role FROM
table1 WHERE code > 0
```

Suppose we also have a table named `table1` that the mapping assertions refer to:

iri	value	code	role
iri1	A	1	P
iri2	B	2	P
iri3	A	2	Q
iri4	B	2	Q

Then `mapping1` will generate the following triples:

```
iri1 a :A_1 .
iri2 a :B_2 .
iri3 a :A_2 .
iri4 a :B_2 .
```

And `mapping2` will generate the following triples:

```
iri1 :P_1 A .
iri2 :P_2 B .
iri3 :Q_2 A .
iri4 :Q_2 B .
```


Community

- People
- Organizations
- Support
 - FAQ
 - Mailing list (Google group)
 - Bug tracking system
 - Commercial support

Contributing

- How to contribute
- Bug report
- Pull request
- Documentation

Support

FAQ

Please visit our pages about [Frequently Asked Questions](#).

Mailing list (Google group)

We have a [mailing list \(Google group\)](#) for anything related to support, discussion and announcements. Please join for any inquiry.

Bug tracking system

To see the list of open issues/bugs, visit [our GitHub tracker](#)

Commercial support

[Ontopic](#) is offering commercial support for Ontop.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Community

- People
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Pull request
- Documentation

How to contribute

Contributions from the community are essential for an open-source project like Ontop and are warmly welcomed.

You can contribute in many differents ways, including:

- Filling a bug report
- Proposing a pull request
- Improving the documentation
- Participating to the mailing list
- Posting about Ontop
- Applying to a job position related to Ontop

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Community

- People
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Reporting a possible bug
- Asking for help or information
- Pull request
- Documentation

Bug report

If you are sure you found a bug, use [GitHub Issues](#) to submit the report; otherwise, start a discussion in the [mailing list](#) .

When reporting a bug, please make sure that you are using [the latest version of Ontop](#) and provide a minimal reproducible example.

Reporting a possible bug

- Perform a search between current and past issues. If it is an open issue, add a comment to the existing issue instead of opening a new one.
- Let us know which version and which setting of Ontop you are using (Protégé, SPARQL endpoint, CLI, Java API)
- State what happened and also state what you expected to see.
- If you have precise steps to reproduce the bug, explain each step and send us a set of ontology/mappings/query (as small as possible) so we can reproduce the problem. Otherwise send us the complete ontology, mapping and query files.
- If you cannot reliably reproduce the test provide details about how often the problem happens and under which conditions it normally happens.
- If the program generated an error, report what the error message was and how it occurred.
- Add the log file in debug mode to help us to track the issue (more information follows).

How to generate the log file

Enable Protégé log

You should go to the folder *conf* of Protégé and modify the line of the *logback.xml* file from `<root Level="info">` to `<root`

`Level="debug">` . When you restart Protégé you will have more information in the command line, that can be useful to understand the problem. You can send us the generated log file from Protégé starting from *Window -> Show log... -> Show log file -> protege.log*

Enable Command Line Interface log

Go to the folder *log* and modify the line of the *logback.xml* file from `<root Level="ERROR">` to `<root Level="debug">` .

Enable logging in the Docker image

Set the environment variable `ONTOP_LOG_LEVEL` to `debug` (`ONTOP_LOG_LEVEL=debug`).

Asking for help or information

See [the support page](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← How to contribute](#)

[Pull request →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Trino

Supported since 5.0.2.

Through the [Trino](#) connector, Ontop is able to construct VKGs on Trino databases.

Limitations & Exceptions

WARNING

Trino does not provide information about integrity constraints. Make sure to provide this information in order to avoid very inefficient queries. We recommend using lenses for this purpose.

- Accessing struct fields with the "dot operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Trino:

```
jdbc.url =
jdbc:trino://localhost:8080/defaultdatabase
jdbc.user = user
jdbc.password = password
jdbc.driver = io.trino.jdbc.TrinoDriver
```

In case Trino is set up without authentication, the `jdbc.user` and `jdbc.password` lines can also be left out.

Nested Type Support

Ontop implements explicit compatibility with the Trino array type `array(t)`. When used with the [flatten lens](#), it is able to automatically infer the type of the result column.

The flatten lens cannot be used on arrays stored as JSON-encoded columns in the database. For such use cases, the column first has to be converted to an array type.

Struct Access

Individual `MAP` structures can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_struct['my_attribute'] FROM ...
```

`ROW` structures can be accessed using the `[position]` operator. `position` must be an integer index starting from 1:

```
SELECT my_struct[1] FROM ...
```

WARNING

Accessing object fields using the "dot operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← TDengine](#)

[FAQ →](#)

Tutorial

Presentation

Basics

Database and Ontop Setup

- Database setup
- Ontop-Protégé setup
- Programme
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Database and Ontop Setup

In this tutorial, we are considering fragments of the information systems of two universities describing students, academic staff and courses.

Database setup

Procedure to set up the database for the following exercises:

1. Unzip the archive of H2 (*h2.zip*)
2. Start the database:
 - On Mac/Linux: open a terminal, go into *h2/bin* and run `h2.sh`
 - On Windows: click on the executable `h2w.bat`
3. After being automatically redirect to the web interface of H2, connect with the default parameters:
 - JDBC URL: `jdbc:h2:tcp://localhost/./university-session1`
 - User name: `sa`
 - No password
4. Now you can see the tables in the schema *uni1*.
5. Try a first SQL query: "Give me the last names of the full professors"

```
SELECT "last_name"
FROM "uni1"."academic"
WHERE "position" = 1
```

Ontop-Protégé setup

Protégé is an open source ontology editor and knowledge management system. Ontop-Protégé is a plugin for designing and testing a VKG specification.

0. Download the latest [Ontop-Protégé bundle](#) for your [Operating System](#) and unzip the archive and go into its folder
1. Run it (*run.bat* on Windows, *run.sh* on Mac/Linux)
2. Register the H2 JDBC driver: go to *Preferences* -> *JDBC Drivers* and add an entry with the following information
 - Description: *h2*
 - Class Name: *org.h2.Driver*
 - Driver file (jar): */path/to/h2/bin/h2-1.4.196.jar*

Programme

1. [Mapping the first data source](#)
2. [Mapping the second data source](#)

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Presentation](#)

[First data source: university 1 →](#)

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Ontology: classes and properties

Mappings

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

First data source: university 1

As a first step, we focus on the database of a first university. It has the schema *uni1*. It is composed of 5 tables.

uni1.student

The table *uni1.student* contains the local ID, first and last names of the students.

s_id	first_name	last_name
1	Mary	Smith
2	John	Doe

The column *s_id* is a primary key.

uni1.academic

Similarly, the table *uni1.academic* contains the local ID, first and last names of the academic staff, but also information about their position.

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

The column *position* is populated with magic numbers:

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

The column *a_id* is a primary key.

uni1.course

The table *uni1.course* contains the local ID and the title of the courses.

c_id	title
1234	Linear Algebra

The column *c_id* is a primary key.

uni1.teaching

The table *uni1.teaching* contains the n-n relation between courses and teachers.

c_id	a_id
1234	1
1234	2

There is no primary key, but two foreign keys to the tables *uni1.course* and *uni1.academic*.

uni1.course-registration

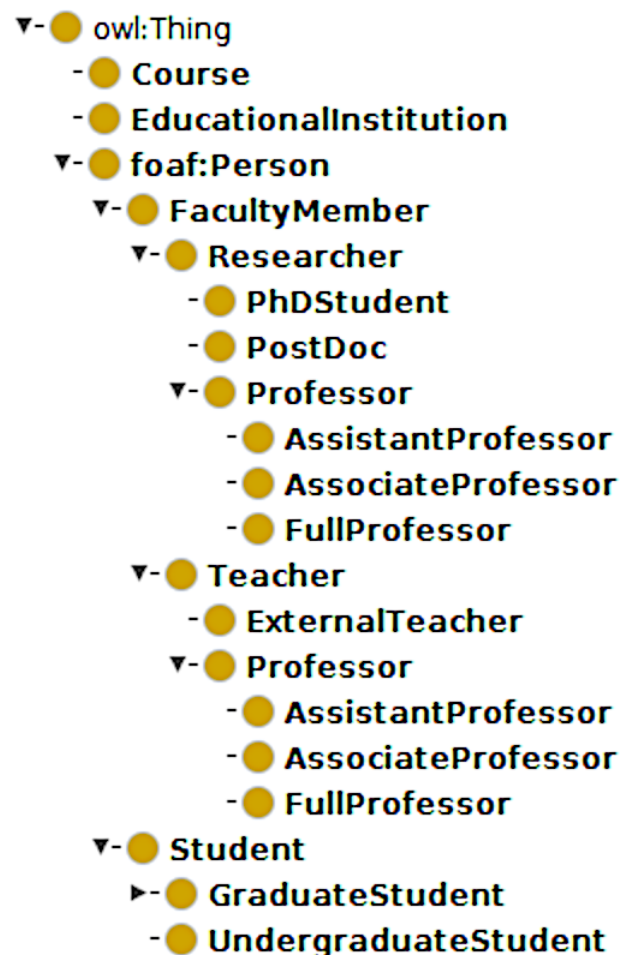
The table *uni1.course-registration* contains the n-n relation between courses and students.

c_id	s_id
1234	1
1234	2

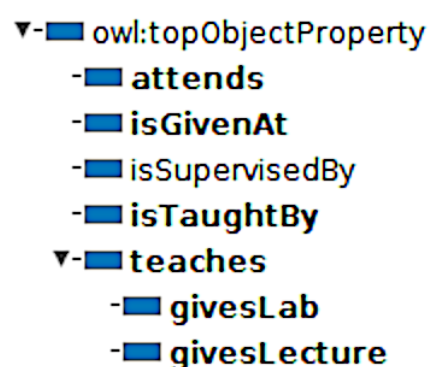
There is no primary key, but two foreign keys to the tables *uni1.course* and *uni1.student*.

Ontology: classes and properties

1. Download [this OWL ontology file](#).
2. Download [this mapping file](#).
3. Download [this properties file](#).
4. In Protégé, go to *File/Open...* to load the ontology file (be sure you have all three files in the same folder).
5. In the tab *Classes* you can visualize the class hierarchy



6. In the tab *Object properties* you can see the properties *attends*, *isGivenAt*, *isSupervisedBy*, *isTaughtBy* and *teaches* (with its two sub-properties *givesLab* and *givesLecture*).



7. In the tab *Data properties* you can see the properties

firstName, lastName and title.

```
▼- owl:topDataProperty
  - foaf:firstName
  - foaf:lastName
  - title
```

Mappings

1. Go to the *Window* -> *Tabs* -> *Ontop mapping* tab
2. Test the already defined connection configuration using the *Test Connection* button
3. Switch to the *Mapping Manager* tab in the ontop mappings tab
4. You should see a first mapping assertion called *uni1-student*
5. Double-click on it to observe it and then close this pop-up window.

Mapping uni1-student

- Target:

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

- Source:

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

Some remarks:

- The target part is described using a *Turtle-like syntax* while the source part is a regular SQL query.
- We used the primary key `s_id` to create the IRI. *As we will see later*, this practice enables Ontop to remove self-joins, which is very important for optimizing the query performance.
- This entry could be split into three mapping assertions


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

Let us now add the other mapping assertions by clicking on *create*:

Mapping uni1-academic

- Target:

```
:uni1/academic/{a_id} a :FacultyMember ;  
    foaf:firstName {first_name}^^xsd:string ;  
    foaf:lastName {last_name}^^xsd:string .
```

- Source:

```
SELECT *  
FROM "uni1"."academic"
```

Mapping uni1-course

- Target:

```
:uni1/course/{c_id} a :Course ;  
    :title {title} ;  
    :isGivenAt :uni1/university .
```

- Source:

```
SELECT *  
FROM "uni1"."course"
```

Mapping uni1-teaching

- Target:

```
:uni1/academic/{a_id} :teaches :uni1/course/{c_id} .
```

- Source:

```
SELECT *  
FROM "uni1"."teaching"
```

Mapping uni1-registration

- Target:

```
:uni1/student/{s_id} :attends :uni1/course/{c_id} .
```

- Source:

```
SELECT *  
FROM "uni1"."course-registration"
```

Mapping uni1-fullProfessor

- Target:

```
:uni1/academic/{a_id} a :FullProfessor .
```

- Source:

```
SELECT *  
FROM "uni1"."academic"  
WHERE "position" = 1
```

Then proceed in a similar way for the other positions (associate professor, assistant professor, external teacher, postdoc).

SPARQL

1. Run Protégé and go to the *Window -> Tabs -> Ontop SPARQL* tab
2. Select Ontop in the *Reasoner* menu
3. Start the reasoner
4. Add a query in the *Query Manager* and run the following query in *SPARQL query editor*:

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

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

Tip: do a right click on the SPARQL query field to visualize the generated SQL query.

Inference

Ontop embeds some inference capabilities and is thus capable of answering a query like the following:

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

SELECT DISTINCT ?teacher {
  ?teacher a :Teacher .
}
```

These inference capabilities can be, for a large part, understood as the ability to infer new mappings from the original mappings and the ontological axioms (e.g. Professor is a sub-class of Teacher). We will discuss it [later in this tutorial](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2
- New mappings
- SPARQL

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Second data source: university 2

We now consider the database of another university. It has a different schema, composed of three tables:

uni2.person

The table *uni2.person* describes the students and the academic staff of the university.

pid	fname	lname	status
1	Zak	Lane	8
2	Mattie	Moses	1
3	Céline	Mendez	2

The column *status* is populated with magic numbers (they differ from the ones of the first university):

- 1 -> Undergraduate Student
- 2 -> Graduate Student
- 3 -> PostDoc
- 7 -> Full Professor
- 8 -> Associate Professor
- 9 -> Assistant Professor

As you can see, undergraduate and graduate students are now distinguished.

uni2.course

The table *uni2.course* contains information about courses, their topic, their lecturer and their lab teacher.

Federation

cid	lecturer	lab_teacher	topic
1	1	3	Information security

`cid` is a primary key. Note that in this data source, there are at most two teachers per course. Lecturers and lab teachers are now distinguished.

uni2.registration

The table *uni2.registration* contains the n-n relation between courses and attendees.

pid	cid
2	1

There is no primary key, but two foreign keys to the tables *uni2.course* and *uni2.person*.

New mappings

Let us add the following mapping assertions together with the previous one.

Mapping uni2.person

- Target:

```
:uni2/person/{pid} a foaf:Person ;
    foaf:firstName {fname}^^xsd:string ;
    foaf:lastName {lname}^^xsd:string .
```

- Source:

```
SELECT *
FROM "uni2"."person"
```

Mapping uni2-course

- Target:

```
:uni2/course/{cid} a :Course ;  
    :title {topic}^^xsd:string ;  
    :isGivenAt :uni2/university .
```

- Source:

```
SELECT *  
FROM "uni2"."course"
```

Mapping uni2-lecturer

- Target:

```
:uni2/person/{lecturer} :givesLecture  
:uni2/course/{cid} .
```

- Source:

```
SELECT *  
FROM "uni2"."course"
```

Mapping uni2-lab-teacher

- Target:

```
:uni2/person/{lab_teacher} :givesLab  
:uni2/course/{cid} .
```

- Source:

```
SELECT *  
FROM "uni2"."course"
```

Mapping uni2-registration

- Target:

```
:uni2/person/{pid} :attends :uni2/course/{cid} .
```

- Source:

```
SELECT *  
FROM "uni2"."registration"
```

Mapping uni2-undergraduate

- Target:

```
:uni2/person/{pid} a :UndergraduateStudent .
```

- Source:

```
SELECT *  
FROM "uni2"."person"  
WHERE "status" = 1
```

And so on for the graduate students, postdocs, full professors, associate professors and assistant professors.

SPARQL

We can now run the previous SPARQL queries and observe that the results combine entries from the two datasets.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

← [First data source: university 1](#) [Deploying a SPARQL endpoint](#) →

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Deploying a SPARQL endpoint

To deploy a SPARQL endpoint, we recommend to use the Ontop endpoint from the

- Command Line Interface or from the
- Docker image

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← Second data source: university 2
Setting up an Ontop SPARQL endpoint with Ontop CLI →

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Setting up an Ontop SPARQL endpoint with Ontop CLI

1. Download [Ontop CLI](#) and unzip it to a directory, which is denoted as `$ONTOP_CLI_DIR` below.
2. Copy the H2 jdbc driver to `$ONTOP_CLI_DIR/jdbc` (if not done yet)
 - Mac/Linux: `cp jdbc/h2-1.4.196.jar $ONTOP_CLI_DIR/jdbc`
3. Start the h2 database.
4. Go to the `endpoint/` directory. Alternatively, if you don't have already the tutorial files, you can download [this OWL ontology file](#), [this mapping file](#), [this properties file](#) and paste them in `input/`.
5. Start the Ontop endpoint. On Mac/Linux:

```
$ONTOP_CLI_DIR/ontop endpoint \
  --ontology=input/university-complete.ttl \
  --mapping=input/university-complete.obda \
  --properties=input/university-
complete.properties \
  --cors-allowed-origins=http://yasgui.org # if
needed
```

On Windows:

```
ontop endpoint ^
  --ontology=input/university-complete.ttl ^
  --mapping=input/university-complete.obda ^
  --properties=input/university-
complete.properties ^
  --cors-allowed-origins=http://yasgui.org
```

6. Open the web interface of the SPARQL endpoint to try some queries: <http://localhost:8080/>

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Deploying a SPARQL endpoint](#)

[Setting up an Ontop SPARQL endpoint with Docker →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker
- How to use this image

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

Setting up an Ontop SPARQL endpoint with Docker

How to use this image

The Docker image `ontop/ontop` is for fast setting up an Ontop SPARQL endpoint. One can either use this image directly, or create a dedicated image based on this image.

1. Go to the `endpoint/` directory. Alternatively, if you don't have already the tutorial files, you can download [this OWL ontology file](#), [this mapping file](#), [this properties file](#) and paste them in `input/`.
2. Make sure to have the `jdbc/` directory and the JDBC driver inside.

In addition, we need the `h2 database` as usual. Assume the `h2 database` is already running.

NB: Linux users have to modify the property `jdbc.url` in `input/university-complete.docker.properties`. Replace `host.docker.internal` with the IP address of your machine (you can see it running the `ifconfig` command).

Use `ontop/ontop` directly

We can start an Ontop SPARQL endpoint by using the `ontop/ontop` image directly. On Linux/Mac:

```
docker run --rm \
  -v $PWD/input:/opt/ontop/input \
  -v $PWD/jdbc:/opt/ontop/jdbc \
  -e
  ONTOP_ONTOLOGY_FILE=/opt/ontop/input/university-
  complete.ttl \
  -e
```

```

ONTOP_MAPPING_FILE=/opt/ontop/input/university-
complete.obda \
    -e
ONTOP_PROPERTIES_FILE=/opt/ontop/input/university-
complete.docker.properties \
    -p 8080:8080 \
    ontop/ontop

```

On Windows:

```

docker run --rm ^
    -v %CD%/input:/opt/ontop/input ^
    -v %CD%/jdbc:/opt/ontop/jdbc ^
    -e
ONTOP_ONTOLOGY_FILE=/opt/ontop/input/university-
complete.ttl ^
    -e
ONTOP_MAPPING_FILE=/opt/ontop/input/university-
complete.obda ^
    -e
ONTOP_PROPERTIES_FILE=/opt/ontop/input/university-
complete.docker.properties ^
    -p 8080:8080 ^
    ontop/ontop

```

Now we can open <http://localhost:8080/> to test SPARQL queries.

Create a dedicated image

In case we want to deploy a self-contained image, we can write a complete `Dockerfile`:

```

FROM ontop/ontop
WORKDIR /opt/ontop
COPY input/university-complete.ttl
input/university-complete.obda input/university-
complete.docker.properties input/
COPY jdbc/h2-1.4.196.jar jdbc/
EXPOSE 8080
ENTRYPOINT java -cp ./lib/*:./jdbc/* -
Dlogback.configurationFile=file:./log/logback.xml
\
    it.unibz.inf.ontop.cli.Ontop endpoint \

```

dockerfile

```
--ontology=input/university-complete.ttl \
--mapping=input/university-complete.obda \
--properties=input/university-
complete.docker.properties \
--cors-allowed-origins=http://yasgui.org \
--lazy # if needed
```

Then, run the commands to build and run the Docker image:

```
$ docker build -t my-ontop-endpoint .
$ docker run -it --rm --name my-running-ontop-
endpoint -p 8080:8080 my-ontop-endpoint
```

Use Docker-compose

Docker-compose allows setting up a number of containers together. For example, the following `docker-compose.yml` file creates a cluster consisting of an H2 database (`db`) and an Ontop SPARQL endpoint (`ontop`).

```
version: '3.4'

services:
  db:
    image: openjdk:8-jdk-alpine
    volumes:
      - ./h2:/opt/h2
    command: [ "java", "-cp", "/opt/h2/bin/h2-
1.4.196.jar", "org.h2.tools.Server", "-
tcpAllowOthers" ]
    ports:
      - "8082:8082"
      - "9082:9082"
  ontop:
    image: ontop/ontop
    environment:
      ONTOP_ONTOLOGY_FILE:
/opt/ontop/input/university-complete.ttl
      ONTOP_MAPPING_FILE:
/opt/ontop/input/university-complete.obda
      ONTOP_PROPERTIES_FILE:
/opt/ontop/input/university-complete.compose.propertie
```

```
    ONTOP_PORTAL_FILE:
/opt/ontop/input/university-complete.portal.toml
    ONTOP_CORS_ALLOWED_ORIGINS: "*"
    ONTOP_DEV_MODE: "true"
    ONTOP_LAZY_INIT: "true"
  volumes:
    - ./input:/opt/ontop/input
    - ./jdbc:/opt/ontop/jdbc
  ports:
    - "8080:8080"
```

Now we can simply start it:

```
$ docker-compose up
```

It exposes the following two ports for the browser:

- <http://localhost:8082> H2 Web Console
- <http://localhost:8080> Ontop SPARQL endpoint

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Interact with an Ontop SPARQL Endpoint

An Ontop Endpoint is accessible by the standard [SPARQL HTTP protocol](#)

URL of the Ontop SPARQL Endpoint

When the endpoint is created by the Ontop CLI or Docker, the URL looks like `http://localhost:8080/sparql`.

Sending HTTP Requests

You can use POST or GET requests carrying the SPARQL query to evaluate (as per SPARQL HTTP protocol).

For example, with POST:

```
POST http://localhost:8080/sparql
Content-Type: application/sparql-query
Accept: application/json

PREFIX : <http://example.org/voc#>
SELECT DISTINCT ?teacher {
  ?teacher a :Teacher .
}
```

Using cURL from the command line

The above request can be sent with the `cURL` command:

```
curl --request POST \
  --url http://localhost:8080/sparql \
  --header 'accept: application/json' \
  --header 'content-type: application/sparql-
```

```
query' \  
  --data 'PREFIX : <http://example.org/voc#>  
  SELECT DISTINCT ?teacher {?teacher a :Teacher .}'
```

Using a SPARQL client library

Alternatively, you may use one of the many [SPARQL clients](#) available for many programming and data analysis environments, as we demonstrate next [using the](#) `SPARQLWrapper` [library within a Python Jupyter Notebook](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Setting up an Ontop SPARQL endpoint with Docker](#)

[Use Jupyter Notebook with an Ontop SPARQL endpoint →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint
- SPARQLWrapper
- Pandas DataFrame

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Use Jupyter Notebook with an Ontop SPARQL endpoint

This tutorial shows how to use a Python/Jupyter Notebook to interact with an Ontop SPARQL endpoint.

Link to the Jupyter Notebook: [ontop-jupyter.ipynb](#)

Assume that the endpoint is already set up and the URL is <http://localhost:8080/sparql> .

SPARQLWrapper

You can use the [SPARQLWrapper library](#) to send SPARQL queries and get results. The following code gets the result as JSON documents and convert it to a Python dict object.

```
from SPARQLWrapper import SPARQLWrapper, JSON

q = """
    PREFIX : <http://example.org/voc#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>

    SELECT ?lname ?title
    WHERE {
        ?attendee foaf:lastName ?lname ; :attends ?
course .
        ?course :title ?title .
    }
    """

sparql =
SPARQLWrapper("http://localhost:8080/sparql")
sparql.setQuery(q)
sparql.setReturnFormat(JSON)
results = sparql.query().convert()
print(results)
```

Federation

Running the code above yields the following output (JSON indentation added for clarity):

```
{
  "head":{
    "vars":[ "lname", "title" ]
  },
  "results":{
    "bindings":[ {
      "lname":{ "type":"literal",
"value":"Robards" },
      "title":{ "type":"literal", "value":"Data
Mining" }
    }, {
      "lname":{ "type":"literal",
"value":"Smith" },
      "title":{ "type":"literal",
"value":"Operating Systems" }
    }, {
      "lname":{ "type":"literal", "value":"Doe"
},
      "title":{ "type":"literal",
"value":"Linear Algebra" }
    }, {
      "lname":{ "type":"literal",
"value":"Combs" },
      "title":{ "type":"literal",
"value":"Operating Systems" }
    }, {
      "lname":{ "type":"literal", "value":"Doe"
},
      "title":{ "type":"literal",
"value":"Analysis" }
    }, {
      "lname":{ "type":"literal",
"value":"Hinkley" },
      "title":{ "type":"literal", "value":"Data
Mining" }
    }, {
      "lname":{ "type":"literal",
"value":"Robards" },
      "title":{ "type":"literal",
"value":"Research Methods" }
    }, {
      "lname":{ "type":"literal",
```

```

"value":"Smith" },
    "title":{" type":"literal",
"value":"Analysis" }
    }, {
    "lname":{" type":"literal",
"value":"Smith" },
    "title":{" type":"literal",
"value":"Linear Algebra" }
    }, {
    "lname":{" type":"literal",
"value":"Combs" },
    "title":{" type":"literal",
"value":"Linear Algebra" }
    }, {
    "lname":{" type":"literal",
"value":"Hinkley" },
    "title":{" type":"literal",
"value":"Theory of Computing" }
    }, {
    "lname":{" type":"literal",
"value":"Alfaro" },
    "title":{" type":"literal",
"value":"Introduction to programming" }
    }, {
    "lname":{" type":"literal",
"value":"Hinkley" },
    "title":{" type":"literal",
"value":"Software factory" }
    }, {
    "lname":{" type":"literal",
"value":"Mendez" },
    "title":{" type":"literal",
"value":"Software factory" }
    }, {
    "lname":{" type":"literal",
"value":"Mendez" },
    "title":{" type":"literal",
"value":"Software process management" }
    }, {
    "lname":{" type":"literal",
"value":"Moses" },
    "title":{" type":"literal",
"value":"Information security" }
    }, {
    "lname":{" type":"literal",
"value":"Moses" },
    "title":{" type":"literal",

```

```
"value":"Discrete mathematics and logic" }
    } ]
  }
}
```

Pandas DataFrame

You might want to convert the SPARQL results to a pandas DataFrame for data analysis. The library `sparql-dataframe` is handy for this.

```
import sparql_dataframe

endpoint = "http://localhost:8080/sparql"

q = """
    PREFIX : <http://example.org/voc#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>

    SELECT ?lname ?course_title
    WHERE {

        ?attendee foaf:lastName ?lname ;
            :attends ?course .
        ?course :title ?course_title .
    }
    """

df = sparql_dataframe.get(endpoint, q)
```

The result `df` can be manipulated as any Pandas DataFrame. For instance:

```
df.head()
```

	lname	course_title
0	Robards	Data Mining
1	Smith	Operating Systems
2	Doe	Linear Algebra

3	Combs	Operating Systems
4	Doe	Analysis

```
df.describe()
```

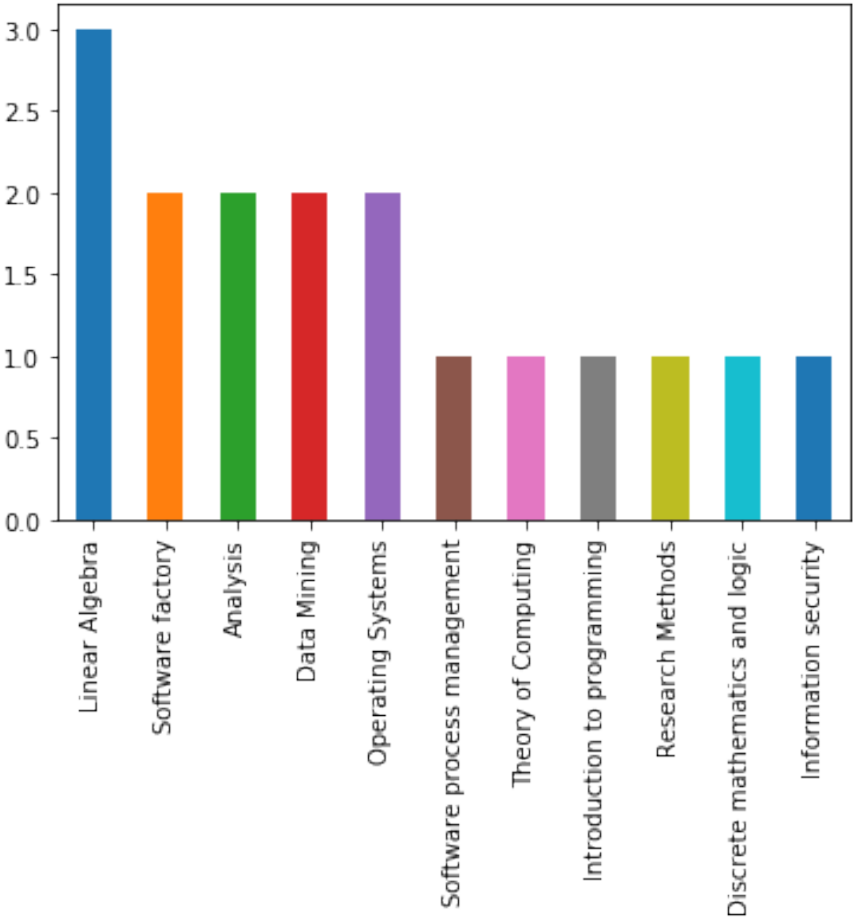
py

	Iname	course_title
count	17	17
unique	8	11
top	Smith	Linear Algebra
freq	3	3

```
df['course_title'].value_counts().plot(kind='bar')
```

py

```
<matplotlib.axes._subplots.AxesSubplot at 0x11475ebd0>
```



[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Interact with an Ontop SPARQL Endpoint](#)

[Second session: Mapping Engineering →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Second session: Mapping Engineering

We discuss here some advanced topics related to mappings engineering and the exploitation of available database constraints, which affect the performance of generated SQL queries:

1. Role of primary keys
2. Role of foreign keys
3. Choice of the IRI templates
4. Existential reasoning

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← Use Jupyter Notebook with an Ontop SPARQL endpoint
Role of primary keys (unique constraints) →

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Role of primary keys (unique constraints)

Unique constraints (such as primary keys) are very useful for removing self-joins and thus improving query answering performance.

Let us now consider the following files: [university-no-pk.ttl](#), [university-no-pk.obda](#) and [university-no-pk.properties](#) files. The mapping assertions are the same as during the first session. The only difference is that primary keys have been removed.

Open the new ontology file in Protégé and run the following SPARQL query:

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

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

Due to the absence of primary keys, the generated SQL is, after some minor reformatting, the following:

```
SELECT *
FROM (

    SELECT 1 AS "profQuestType", NULL AS
"profLang",
          ('http://example.org/uni2/person/' ||
QVIEW1."pid") AS "prof",
          7 AS "lastNameQuestType", NULL AS
"lastNameLang",
          QVIEW3."lname" AS "lastName"
```

```

FROM    "uni2"."person" QVIEW1,
        "uni2"."person" QVIEW2,
        "uni2"."person" QVIEW3
WHERE   (QVIEW1."status" = 7) AND
        QVIEW1."pid" IS NOT NULL AND
        (QVIEW1."pid" = QVIEW2."pid") AND
        QVIEW2."fname" IS NOT NULL AND
        (QVIEW1."pid" = QVIEW3."pid") AND
        QVIEW3."lname" IS NOT NULL

UNION

SELECT  1 AS "profQuestType", NULL AS
"profLang",
        ('http://example.org/uni1/academic/' ||
QVIEW1."a_id") AS "prof",
        7 AS "lastNameQuestType", NULL AS
"lastNameLang",
        QVIEW3."last_name" AS "lastName"
FROM    "uni1"."academic" QVIEW1,
        "uni1"."academic" QVIEW2,
        "uni1"."academic" QVIEW3
WHERE   (QVIEW1."position" = 1) AND
        QVIEW1."a_id" IS NOT NULL AND
        (QVIEW1."a_id" = QVIEW2."a_id") AND
        QVIEW2."first_name" IS NOT NULL AND
        (QVIEW1."a_id" = QVIEW3."a_id") AND
        QVIEW3."last_name" IS NOT NULL

) SUB_QVIEW

```

In each sub-query, one can observe two self-joins, between

`QVIEW1`, `QVIEW2` and `QVIEW3`.

If you run the same query with the setting of the first session, you will obtain the following query:

```

SELECT  1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni2/person/' ||
QVIEW1."pid" ) AS "p",
        7 AS "lastNameQuestType", NULL AS
"lastNameLang",
        QVIEW1."lname" AS "lastName"
FROM    "uni2"."person" QVIEW1
WHERE   QVIEW1."pid" IS NOT NULL AND

```

```

QVIEW1."fname" IS NOT NULL AND
QVIEW1."lname" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni1/academic/' ||
QVIEW1."a_id" ) AS "p",
      7 AS "firstNameQuestType", NULL AS
"firstNameLang",
      QVIEW1."first_name" AS "firstName",
      7 AS "lastNameQuestType", NULL AS
"lastNameLang",
      QVIEW1."last_name" AS "lastName"
FROM   "uni1"."academic" QVIEW1
WHERE  QVIEW1."a_id" IS NOT NULL AND
       QVIEW1."first_name" IS NOT NULL AND
       QVIEW1."last_name" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni1/student/' ||
QVIEW1."s_id" ) AS "p",
      7 AS "firstNameQuestType", NULL AS
"firstNameLang",
      QVIEW1."first_name" AS "firstName",
      7 AS "lastNameQuestType", NULL AS
"lastNameLang",
      QVIEW1."last_name" AS "lastName"
FROM   "uni1"."student" QVIEW1
WHERE  QVIEW1."s_id" IS NOT NULL AND
       QVIEW1."first_name" IS NOT NULL AND
       QVIEW1."last_name" IS NOT NULL

```

As you can see, the self-joins are removed when primary keys are provided and used as joining conditions.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Second session: Mapping Engineering](#) [Role of foreign keys →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Role of foreign keys

Foreign keys play a crucial role for optimizing the saturated mapping assertions by doing some query containment checks.

Let us now consider the case where foreign keys are missing. Please download the following files: [university-no-fk.ttl](#), [university-no-fk.obda](#) and [university-no-fk.properties](#) files.

Let us consider the case of `foaf:Person`. If you run the following SPARQL query in the absence of foreign keys:

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

SELECT ?p {
  ?p a foaf:Person .
}
```

you will obtain a SQL query similar to the following one (after some minor reformatting):

```
SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni2/person/' ||
QVIEW1."lab_teacher") AS "p"
FROM   "uni2"."course" QVIEW1
WHERE  QVIEW1."lab_teacher" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni2/person/' ||
QVIEW1."lecturer") AS "p"
FROM   "uni2"."course" QVIEW1
WHERE  QVIEW1."lecturer" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
```

```

        ('http://example.org/uni1/academic/' ||
QVIEW1."a_id") AS "p"
FROM    "uni1"."teaching" QVIEW1
WHERE   QVIEW1."a_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni2/person/' ||
QVIEW1."pid") AS "p"
FROM    "uni2"."registration" QVIEW1
WHERE   QVIEW1."pid" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni1/student/' ||
QVIEW1."s_id") AS "p"
FROM    "uni1"."course-registration" QVIEW1
WHERE   QVIEW1."s_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni1/student/' ||
QVIEW1."s_id") AS "p"
FROM    "uni1"."student" QVIEW1
WHERE   QVIEW1."s_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni1/academic/' ||
QVIEW1."a_id") AS "p"
FROM    "uni1"."academic" QVIEW1
WHERE   QVIEW1."a_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
        ('http://example.org/uni2/person/' ||
QVIEW1."pid") AS "p"
FROM    "uni2"."person" QVIEW1
WHERE   QVIEW1."pid" IS NOT NULL

```

This SQL query is the union of eight sub-queries. Basically, it

queries the tables `uni1.student`, `uni1.academic`, `uni2.person` as expected, but also `uni1.teaching`, `uni1.course-registration`, `uni2.course` and `uni2.registration`. Recall that the presence of the four latter tables is due to the fact that, according to the ontology, the respective domains of the properties `:givesLecture`, `:givesLab` and `:attends` are subsumed by `foaf:Person`.

With the setting of the first session which includes foreign keys, the generated SQL query contains only three unions:

```
SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni1/student/' ||
QVIEW1."s_id") AS "p"
FROM   "uni1"."student" QVIEW1
WHERE  QVIEW1."s_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni1/academic/' ||
QVIEW1."a_id") AS "p"
FROM   "uni1"."academic" QVIEW1
WHERE  QVIEW1."a_id" IS NOT NULL

UNION ALL

SELECT 1 AS "pQuestType", NULL AS "pLang",
      ('http://example.org/uni2/person/' ||
QVIEW1."pid") AS "p"
FROM   "uni2"."person" QVIEW1
WHERE  QVIEW1."pid" IS NOT NULL
```

Side note

If we now consider also the first and the last names of persons, foreign keys are not needed to optimize the query.

Try:

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



```
SELECT ?p ?firstName ?lastName {
  ?p a foaf:Person ;
    foaf:firstName ?firstName ;
    foaf:lastName ?lastName .
}
```

and observe that the query produces only three unions.

Why? Because the domain of `foaf:firstName` and `foaf:lastName` is `foaf:Person`. The SPARQL query can thus be safely rewritten as follow:

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

SELECT ?p ?firstName ?lastName {
  ?p foaf:firstName ?firstName ;
    foaf:lastName ?lastName .
}
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Role of primary keys \(unique constraints\)](#)

[Choice of the IRI templates →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
 - Manual approach
 - Canonical IRIs
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Choice of the IRI templates

The choice of IRI templates may impact query complexity and performance, depending on whether joins have to be introduced to materialize those templates.

Until now, we have been using local identifiers from the data sources to build three different IRI templates for the persons of the two universities:

- `ex:uni1/student/{*}`
- `ex:uni1/academic/{*}`
- `ex:uni2/person/{*}`

Let us now consider the case where the tables `uni1.student`, `uni1.academic` and `uni2.person` have a new column: `ssn`. This column corresponds to the social security number of the person.

This new column gives us the opportunity to use a common IRI template for all the persons. Such IRI template would allow us to collect information about a person registered in both universities.

As we will see, there are actually some persons that are teaching or studying in both universities.

Given that we already have the mapping assertions produced during the first session, we will consider two ways to change the IRI templates: first by doing it manually, then using the notion of *canonical IRIs*.

We will also see that using this new IRI template has a negative impact on the performance of some queries.

Manual approach

Download the following files:

Federation

- [university-manual-ssn.ttl](#)
- [university-manual-ssn.obda](#)
- [university-manual-ssn.properties](#)

The mapping assertions are currently the same as during the first hands-on session. Let us now update them with the new IRI template for persons.

Mapping uni1.student

- Target:

```
ex:person/{ssn} a :Student ;
    foaf:firstName {first_name}^^xsd:string ;
    foaf:lastName {last_name}^^xsd:string .
```

- Source:

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

Mapping uni1-academic

- Target:

```
ex:person/{ssn} a :FacultyMember ;
    foaf:firstName {first_name}^^xsd:string ;
    foaf:lastName {last_name}^^xsd:string .
```

- Source:

```
SELECT *
FROM "uni1"."academic"
```

Mapping uni1-teaching

- Target:

```
ex:person/{ssn} :teaches ex:uni1/course/{c_id} .
```

- Source:

```
SELECT ac."ssn", te."c_id"
FROM "uni1"."teaching" te, "uni1"."academic" ac
WHERE te."a_id" = ac."a_id"
```

As you can see, a join is needed to get the SSN of the teacher.

Mapping uni1-registration

- Target:

```
ex:person/{ssn} :attends ex:uni1/course/{c_id} .
```

- Source:

```
SELECT st."ssn", re."c_id"
FROM "uni1"."course-registration" re,
"uni1"."student" st
WHERE re."s_id" = st."s_id"
```

Mapping uni1-fullProfessor

- Target:

```
ex:person/{ssn} a :FullProfessor .
```

- Source:

```
SELECT *
FROM "uni1"."academic"
WHERE "position" = 1
```

Then proceed in a similar way for the other positions (assistant professor, postdoc, etc.).

Mapping uni2.person

- Target:

```
ex:person/{ssn} a foaf:Person ;
    foaf:firstName {fname}^^xsd:string ;
    foaf:lastName {lname}^^xsd:string .
```

- Source:

```
SELECT *
FROM "uni2"."person"
```

Mapping uni2-lecturer

- Target:

```
ex:person/{ssn} :givesLecture ex:uni2/course/{cid}
.
```

- Source:

```
SELECT pe."ssn", co."cid"
FROM "uni2"."course" co, "uni2"."person" pe
WHERE co."lecturer" = pe."pid"
```

Mapping uni2-lab-teacher

- Target:

```
ex:person/{ssn} :givesLab ex:uni2/course/{cid} .
```

- Source:

```
SELECT pe."ssn", co."cid"
FROM "uni2"."course" co, "uni2"."person" pe
```

```
WHERE co."lab_teacher" = pe."pid"
```

Mapping uni2-registration

- Target:

```
ex:person/{ssn} :attends ex:uni2/course/{cid} .
```

- Source:

```
SELECT pe."ssn", re."cid"
FROM "uni2"."registration" re, "uni2"."person" pe
WHERE re."pid" = pe."pid"
```

Mapping uni2-undergraduate

- Target:

```
ex:person/{ssn} a :UndergraduateStudent .
```

- Source:

```
SELECT *
FROM "uni2"."person"
WHERE "status" = 1
```

Then proceed in a similar way for the other positions (assistant professor, postdoc, etc.).

SPARQL queries

With the new mapping assertions, the following SPARQL query should now return some results:

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

```
SELECT DISTINCT ?firstName ?lastName ?ins1 ?ins2 {
  ?p foaf:firstName ?firstName ;
    foaf:lastName ?lastName ;
    :teaches [ :isGivenAt ?ins1 ],
              [ :isGivenAt ?ins2 ] .

  FILTER(str(?ins1) < str(?ins2))
}
```

However, if you execute another variant, you will see it is now very inefficient:

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

SELECT ?firstName ?lastName ?ins1 ?ins2 {
  ?p1 foaf:firstName ?firstName ;
    foaf:lastName ?lastName ;
    :teaches [ :isGivenAt ?ins1 ] .

  ?p2 foaf:firstName ?firstName ;
    foaf:lastName ?lastName ;
    :teaches [ :isGivenAt ?ins2 ] .

  FILTER(str(?ins1) < str(?ins2))
}
```

Look at the SQL query, it should be very long as it suffers from a substantial exponential blowup. Note that the first query also suffers from an exponential blowup but the latter is less important and thus less sensible on our small dataset.

Canonical IRIs

Instead of changing the mapping assertions manually, Ontop allows you to declare a canonical IRI template for

```
ex:uni1/student/{*} , ex:uni1/academic/{*} and
ex:uni2/person/{*} .
```

This takes the form of three additional mapping assertions that can be found in the [university-canonical.ttl](#), [university-](#)

[canonical.obda](#), and [university-canonical.properties](#) files.

Mapping uni1-academic-canonical

- Target:

```
:person/{ssn} obda:isCanonicalIRIOf  
:uni1/academic/{a_id} .
```

- Source:

```
SELECT * FROM "uni1"."academic"
```

Mapping uni1.student-canonical

- Target:

```
:person/{ssn} obda:isCanonicalIRIOf  
:uni1/student/{s_id} .
```

- Source:

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

Mapping uni2.person-canonical

- Target:

```
:person/{ssn} obda:isCanonicalIRIOf  
:uni2/person/{pid} .
```

- Source:

```
SELECT * FROM "uni2"."person"
```

With these three mapping assertions and the ones of the first

session, Ontop produces the same saturated mapping assertions as in the manual approach. Query evaluation performance is thus the same (and similarly affected to the additional joins needed to build person IRIs), but the use of canonical IRI templates makes simpler for users to define and adapt mappings.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Role of foreign keys](#)

[Bonus: existential reasoning →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Bonus: existential reasoning

Ontop includes a *tree-witness query rewriting algorithm* implementing *existential reasoning*.

To enable it, in Protégé go to *Preferences* -> *Ontop reasoner* tab and enable the option *reasoning over anonymous individuals*.

Then, restart the reasoner and try the following query that looks for people supervised by some *Professor*:

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

SELECT ?x {
  ?x :isSupervisedBy [ a :Professor ] .
}
```

Note that there is no mapping assertion for the property *isSupervisedBy*. However, the ontology states that for every *GraduateStudent* there must exist some *Professor* so that the student *isSupervisedBy* the professor. Ontop can thus infer that all *GraduateStudent* are answers to the query, which is then rewritten as the following equivalent query prior to evaluation:

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

SELECT ?x {
  ?x a :GraduateStudent .
}
```

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

[How to materialize data into a graph database →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database
- Solutions
- Choosing the Right Approach for Your Use Case

How to materialize data into a graph database

In this section, we present two ways to materialize your Knowledge Graph using Ontop.

Solutions

1. Materialize in RDF files and load into a triplestore

For the first solution, you will need the following prerequisites:

- Access to the tutorial H2 database
- The `mapping file` `university-complete.obda` and the `properties file` `university-complete.properties`
- `Ontop CLI`

Using the CLI command `ontop-materialize` (<https://ontop-vkg.org/guide/cli#ontop-materialize>), you can `materialize` your KG into one or multiple files. For simplicity, we keep the default option and only materialize it into one file.

```
sh
./ontop materialize
--mapping=university-complete.obda \
--properties=university-complete.properties \
-f turtle \
-o materialized-triples.ttl
```

After running the command, we have all the content of our KG copied to the file `materialized-triples.ttl`.

Now we load this file in the triplestore of our choice, in this case, we use `GraphDB`. This graph database offers `several ways to load files`. Here, since our file is small, we go for the simplest option

and load it directly from the UI.

Once this is done, we can query this KG using GraphDB.

2. Deploy a VKG and fetch its content from the graph database

For the second solution, we make use of the concept of KG virtualization.

We deploy the KG as a virtual KG first and then query it from the graph database. In this way, you can retrieve the triples and store them locally in the graph database.

Triples are directly streamed to the graph database: no intermediate file storage is involved, making this solution more direct than the previous one.

let's deploy the KG as a virtual KG using the *ontop-endpoint* command:

```
sh
./ontop endpoint
--mapping=input/university-complete.obda \
--properties=input/university-complete.properties \
```

Now Ontop is deployed as a [SPARQL endpoint](http://localhost:8080/sparql) available at <http://localhost:8080/sparql>.

Let's go now to GraphDB. To fetch and insert all the triples from the VKG exposed by Ontop, we run the following SPARQL INSERT query from GraphDB itself:

```
INSERT {
  ?s ?p ?o
}
WHERE {
  SERVICE <http://localhost:8080/sparql> {
    ?s ?p ?o
  }
}
```

This query materializes the same triples as with the first approach.

Choosing the Right Approach for Your Use Case

1. Small Dataset, Easy Communication: If your dataset isn't large and you can easily set up communication between the Ontop SPARQL endpoint and the graph database, go with solution #2. It avoids dealing with files and intermediate storage.
2. Large Dataset, Efficient Loading: For very large datasets, choose the most efficient loading solution supported by the triplestore, even if it requires more effort to set up.
3. Materializing Fragments of the KG: Solution #2 allows easy materialization of specific fragments of the Knowledge Graph by adapting the SPARQL query. You can have hybrid KGs with some parts stored in the graph database and the rest kept virtual.
4. Advantage of Keeping Data Virtual: Keeping data virtual is great for large volumes of sensor data that constantly update. It's better to keep this part virtual while storing rich contextual information in the graph database.

Ontology Usage

If you're familiar with Ontop, you might have noticed that we didn't use an ontology in this example. Providing an ontology to Ontop can result in a significantly larger KG due to the reasoning capabilities embedded in Ontop. However, GraphDB also has reasoning capabilities, allowing reasoning to be done later in GraphDB, making materialization simpler and faster. If your graph database doesn't support reasoning, Ontop can handle it.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Federating multiple databases

- Ontop with Dremio
- Ontop with Denodo
- Ontop with Teiid

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← [How to materialize data into a graph database](#) [Ontop with Denodo](#) →

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Ontop with Dremio

In this tutorial we present step-by-step the way of connecting Dremio to Ontop. We show how to integrate *uni1* data saved in the PostgreSQL database and *uni2* data saved in plain JSON files into one Dremio data space.

We provide a Docker-compose script, which is both launching Dremio and the PostgreSQL database. Go at the root of the [ontop-tutorial git repository](#) and run the following commands:

```
cd federation/dremio
docker-compose pull && docker-compose up --build
```

sh

Once Dremio is initialized, visit <http://localhost:9047>. Before you continue, we recommend you to see the following tutorials provided by Dremio:

1. [Getting oriented with Dremio](#)
2. [Working with your first dataset](#)

By following the instructions in [Working with your first dataset](#), we create a *space* named university as shown below.

Add and save the new data space:

Spaces »

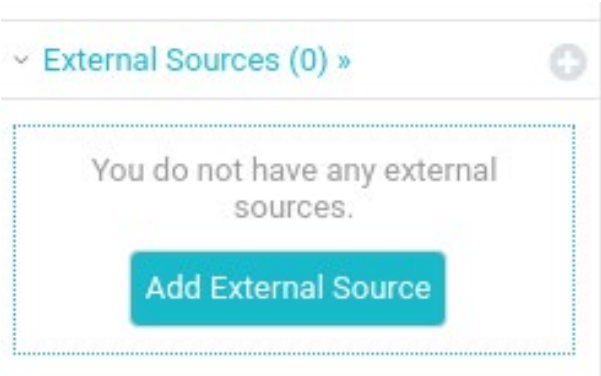


You do not have any spaces.

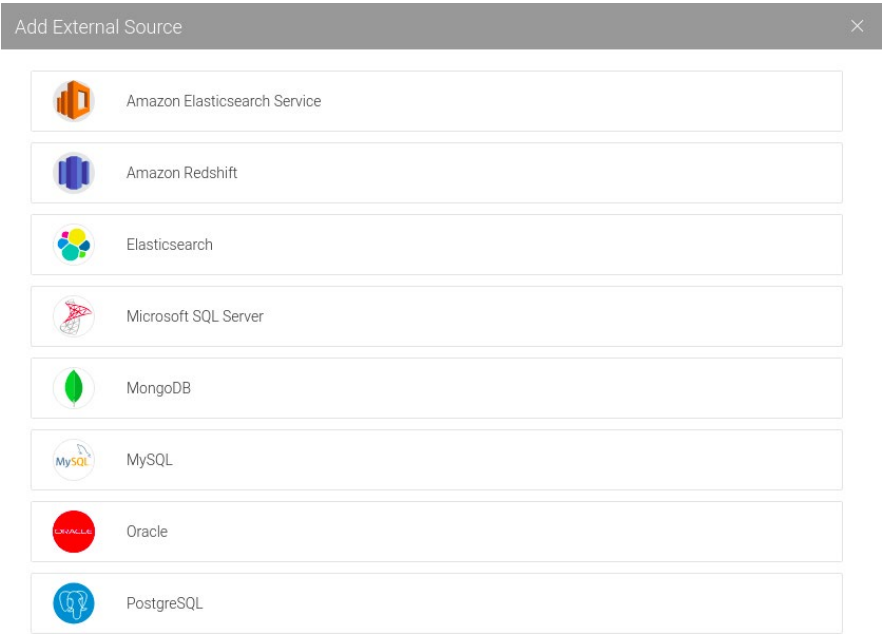
Add Space

It will be our data space in which we integrate data from various sources.

The *uni1* data is contained in a PostgreSQL database named *university-session1*. We are ready to add our database as a new datasource into Dremio:



Select PostgreSQL:



Enter the required JDBC information (password: *postgres*):

New PostgreSQL Source

General

Advanced Options

Reflection Refresh

Metadata

PostgreSQL Source

Name

university-session1

Connection

Host

db

Port

5432

Database Name

university-session1

Encrypt connection

Authentication

No Authentication

Master Credentials

Secret Resource Uri

Username

postgres

Password

Cancel

Save

Now we see the tables in *university-session1*:

university-session1.uni1

Search...

Name	Jobs	Action
academic	—	
course	—	
course-registration	—	
student	—	
teaching	—	

We add the table *course-registration* as dataset into *university*.
We rename it to *uni1-registration*.

Save Dataset As

Name

uni1-registration

Location

@ontopuser

university

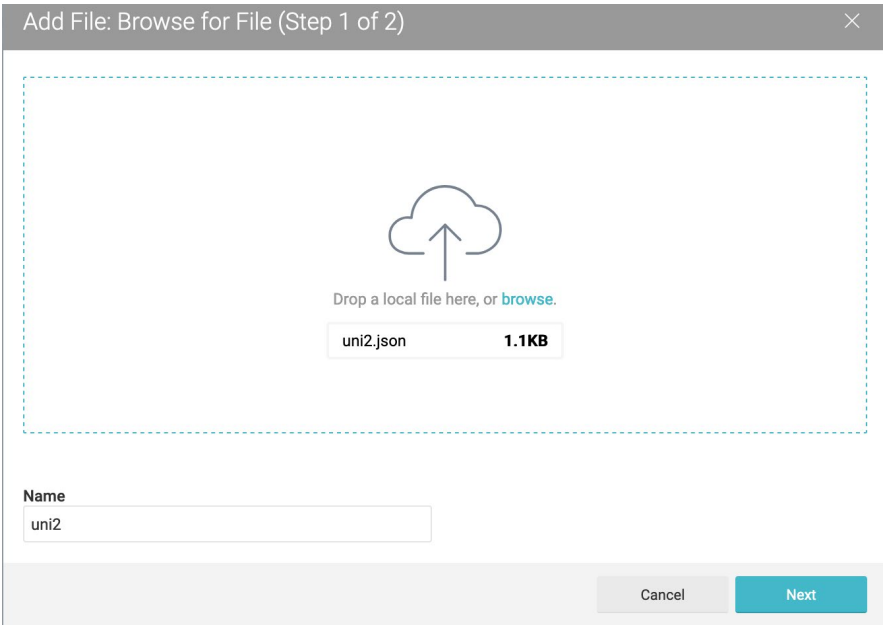
Cancel

Save

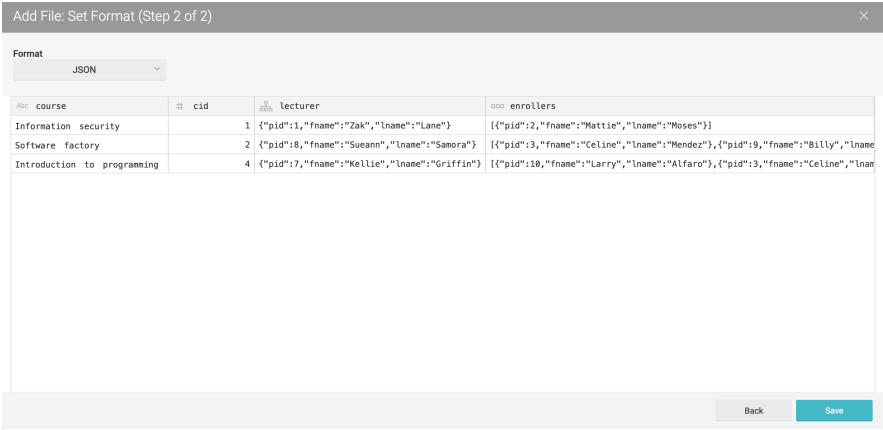
Save the other datasets in similar manner.

Now we add the *uni2* data from [here](#) as a JSON data source:

<https://ontop-vkg.org/tutorial/federation/dremio/>[2026/2/26 18:51:36]



The *uni2* JSON data can be seen as follows:



JSON files usually contain nested data. However, Ontop cannot directly query nested data. For this reason, in order to make our JSON data queryable by Ontop, first we need to extract relevant group of elements, and save these groups as datasets.

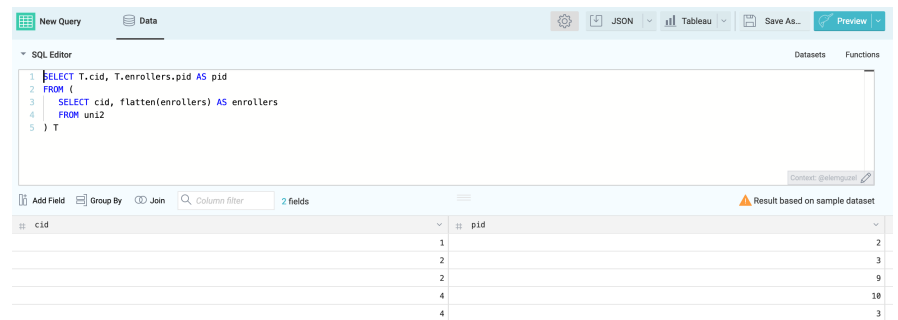
With the following SQL query we create an *uni2-registration* dataset and save it into the data space *university*:

```
SELECT T.cid, T.enrollers.pid AS pid
FROM (
  SELECT cid, flatten(enrollers) AS enrollers
  FROM uni2
) T
```

The following SQL expression flattens the array of enrollers:

```
flatten(enrollers) AS enrollers
```

Save As the data set with the name *uni2-registration*.



The screenshot shows the Dremio SQL Editor interface. The SQL Editor pane contains the following query:

```
1 SELECT T.cid, T.enrollers.pid AS pid
2 FROM (
3   SELECT cid, flatten(enrollers) AS enrollers
4   FROM uni2
5 ) T
```

Below the editor, the results pane shows a table with two columns: **cid** and **pid**. The table contains 5 rows of data. A warning icon and text "Result based on sample dataset" are visible in the top right of the results pane.

cid	pid
1	2
2	3
2	9
4	18
4	3

Create a dataset named *uni2-course* in a similar manner:

```
SELECT course, cid FROM uni2
```

Create a *uni2-student* dataset with a bit more involved SQL that flattens the array of students:

```
SELECT T.enrollers."pid" AS
pid,T.enrollers."fname" AS fname,
T.enrollers."lname" AS lname
FROM(
    SELECT flatten(enrollers) AS enrollers
    FROM uni2
) T
```

Create *uni2-teaching* dataset with the following SQL:

```
SELECT cid, uni2.lecturer.pid AS pid
FROM uni2
```

Now we can list all the datasets we saved in the *university* space:

university			Search...		
Name		Jobs	Action		
uni1-academic		-			
uni1-course		-			
uni1-registration		-			
uni1-student		-			
uni2-course		-			
uni2-registration		1			
uni2-student		-			
uni2-teaching		-			

Finally we are ready to connect Dremio to Ontop. Dremio can be connected to Ontop through its JDBC interface. By following the instructions provided in [here](#) , we provide to Ontop the following JDBC connection information in a ".properties file" for a Dremio instance running on the localhost:

```
jdbc.url=jdbc\:dremio\:direct\=localhost\:31010
jdbc.driver=com.dremio.jdbc.Driver
jdbc.user=dremiotest
jdbc.password=dremiotest
```

Dremio JDBC driver can be downloaded from [here](#) .

Over an OBDA setting containing the following mapping assertions:

```
[PrefixDeclaration]
:      http://example.org/voc#
ex:    http://example.org/
owl:   http://www.w3.org/2002/07/owl#
rdf:   http://www.w3.org/1999/02/22-rdf-syntax-ns#
xml:   http://www.w3.org/XML/1998/namespace
xsd:   http://www.w3.org/2001/XMLSchema#
foaf:  http://xmlns.com/foaf/0.1/
obda:  https://w3id.org/obda/vocabulary#
rdfs:  http://www.w3.org/2000/01/rdf-schema#

[MappingDeclaration] @collection [[
mappingId    uni1-student
target       :uni1/student/{s_id} a :Student ;
foaf:firstName {first_name}^^xsd:string ;
foaf:lastName {last_name}^^xsd:string .
source       SELECT * FROM "university"."uni1-
```

```
student"

mappingId    uni1-attends
target       :uni1/student/{s_id} :attends
              :uni1/course/{c_id}/{title} .
source       SELECT "uni1-registration".s_id,
                    "uni1-course".c_id, "uni1-course".title FROM
                    "university"."uni1-registration",
                    "university"."uni1-course" WHERE "uni1-
                    registration".c_id = "uni1-course".c_id

mappingId    uni2-student
target       :uni2/student/{pid} a :Student ;
foaf:firstName {fname}^^xsd:string ; foaf:lastName
{lname}^^xsd:string .
source       SELECT * FROM "university"."uni2-
student"

mappingId    uni2-attends
target       :uni2/student/{pid} :attends
              :uni2/course/{cid}/{course} .
source       SELECT "uni2-registration".pid, "uni2-
course".cid, "uni2-course".course FROM
"university"."uni2-registration",
"university"."uni2-course" WHERE "uni2-
registration".cid = "uni2-course".cid
]]
```

Now we can execute the following SPARQL query on Ontop:

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

SELECT ?course ?firstName ?lastName {
  ?student :attends ?course .
  ?student foaf:firstName ?firstName .
  ?student foaf:lastName ?lastName .
}
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Ontop with Denodo

NB: You might also consult the following presentation, which shows screenshots for some of the steps described below:
<https://github.com/ontop/ontop-examples/blob/master/cikm-2018-tutorial/4-obdi-demo.pdf>

1. Set up Denodo.

Download Denodo Express

Download it from <https://www.denodo.com/en/denodo-platform/denodo-express>

(register, download the platform and the licence file)

Unzip the file and run the installer.

Note: environment variable `JAVA_HOME` must be set

Running Denodo

```
<DenodoInstallationPath>/bin/denodoPlatform.sh
```

- Select the 'Virtual DataPort' tab
- Start 'Virtual DataPort Server'
- Press the 'Launch' button
- Login with:
 - . Login: `admin`
 - . Password: `admin`
 - . Server: `//localhost:9999/admin`

2. Configure datasets with Denodo

NB: instead of doing the following step by step, you can also load [this SQL script](#) directly within Denodo.

Create a database

We will create two data sources based on web APIs.

First, a Web API with weather data.

- Administration -> Database Management -> New
- Name the database `bzopendata` for instance (leave the rest to default values), and click on the 'OK' button
- In the left window, right-click on the bzopendata database -> New -> Data Source -> JSON
- In the field 'Name', enter `stations`
- In the field 'Data route', select 'HTTP Client'
- Click on the 'Configure' button to enter the URL of the source:
<http://daten.buergernetz.bz.it/services/meteo/v1/stations>
- Click on 'OK', and then press the 'Save' button to save the data source

Then add a second data source for sensor data, repeating all the operations above, but:

- in the field 'Name', enter `sensors`
- use the URL:
<http://daten.buergernetz.bz.it/services/meteo/v1/sensors>

3. Configure the datasources

Select a data source, and click on 'Create base view'

- for stations: use the default "tuple root" (`/JSONFile`)
- for sensors: use the "tuple root": `/JSONFile/JSONArray` (uncheck the checkbox 'JSON root')

Click on 'Save'.

For stations, click on the 'Execution panel' button, and then the button 'Execute'

We get only one result --> We need to flatten the data:

- In the left window, right-click on {stations} -> New -> Flatten (to display the data sources, you may need to right-click on

'bzopendata' -> 'Refresh')

- Right click on the row 'features' in the table -> 'Flatten array features'

In addition, we want to project the subfields of the JSON object 'properties':

- Select the 'Output' tab
- Right-click on the row 'properties' in the table -> 'Project subfields of properties'

Click on 'Save': this creates a new view 'f_stations'

4. Configure Ontop-protege to use a Denodo datasource

- Install the Denodo JDBC driver: Within Protege: File -> Preferences -> 'JDBC Drivers' tab -> Add

. Description: `Denodo`

. Class name: `com.denodo.vdp.jdbc.Driver`

. Driver File (jar): browse to

`<DenodoInstallationPath>/tools/client-drivers/jdbc/denodo-vdp-jdbcdriver.jar`

- For the connection, in your Datasource manager, use:

. Connection url: `jdbc:vdb://localhost:9999/bzopendata`

. Database Username: `admin`

. Database Password: `admin`

. Driver class: `com.denodo.vdp.jdbc.Driver`

5. Create the ontology and mapping and try some SPARQL queries

You can use directly the files we have prepared:

- [bzweather.obda](#)
- [bzweather.owl](#)
- [bzweather.properties](#)
- [bzweather.q](#)

6. Setup a SPARQL endpoint with Command Line Interface

Configure Ontop CLI

Add the JDBC denodo driver for the Ontop CLI:

```
cp <DenodoInstallationPath>/tools/client-
drivers/jdbc/denodo-vdp-jdbcdriver.jar
<OntopInstallationPath>/jdbc
```

Run the Ontop CLI:

```
ontop endpoint -t bzweather.owl -m bzweather.obda
-p bzweather.properties --cors-allowed-origins='*'
```

Access the SPARQL endpoint

Open <http://localhost:8080/>

Run an example SPARQL query:

```
PREFIX geo:
<http://www.opengis.net/ont/geosparql#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX : <http://example.org/weather#>
SELECT *
{
    ?station a :WeatherStation ; geo:asWKT ?wkt;
    :hasSensor ?sensor ; :longitude ?lon; :latitude ?
lat .
    ?sensor sosa:madeObservation ?o .
    ?o sosa:hasSimpleResult ?wktLabel .
    ?o sosa:resultTime ?t .
}
LIMIT 100
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Basic Lens

Basic lenses can be used on one base relation, over which we can apply a filter, an extended projection, and additional constraints.

Projection

For this section, we first look at the table `museums` from the DuckDB database. This table has the following schema:

column	type
<code>museum_id</code>	integer
<code>name</code>	string
<code>address</code>	string
<code>yearly_income</code>	integer
<code>yearly_spendings</code>	integer
<code>ratings</code>	array of floats

The column `museum_id` is a primary key.

Now, we make two decisions:

- The `ratings` column is not important to us, so we want to remove it.
- The columns `yearly_income` and `yearly_spendings` are not interesting, but we would like to know the yearly profit (*income - spendings*)

Both of these goals can be achieved using a single basic lens. The basic lens has the following structure:

```
{
  "name": [String],
}
```

json

```

    "baseRelation": [String],
    "columns": {
      "added": [{
        "name": String,
        "expression": String
      }],
      "hidden": [String]
    },
    "filterExpression": String,
    "type": "BasicLens"
  }

```

We can prepare our `lenses.json` file like this:

```

{
  "relations": [
    {
      "name": ["lenses",
"museum_projection"],
      "baseRelation": ["museums"],
      "columns": {
        "added": ...,
        "hidden": ...
      },
      "type": "BasicLens"
    }
  ]
}

```

As we do not require a filter, we do not need the field `filterExpression`. The field `columns` can be used to list existing columns that should be removed and new columns that should be added.

Above, we decided that the fields `ratings`, `yearly_income` and `yearly_spendings` are not interesting to us. We can easily hide them by including their names in the `hidden` list.

To add a new `yearly_profit` field, we have to add one entry to the `added` list, using `yearly profit` as its name, and using the expression `yearly_income - yearly_spendings` in its `expression` field.

NOTE

Even though we have decided to hide the columns `yearly_income` and `yearly_spendings`, we can still use them for expressions within the same lens.

After making these changes, the full `lenses.json` file should look like this:

```

{
  "relations": [
    {
      "name": ["lenses",
"museum_projection"],
      "baseRelation": ["museums"],
      "columns": {
        "added": [
          {
            "name": "yearly_profit",
            "expression":
"CAST(yearly_income - yearly_spendings as
INTEGER)"
          }
        ],
        "hidden": ["ratings",
"yearly_income", "yearly_spendings"]
      },
      "type": "BasicLens"
    }
  ]
}
    
```

NOTE

In the expression for `yearly_profit`, we cast the result of the subtraction to an `INTEGER`. This way, Ontop is guaranteed to know that the column will be of type `INTEGER`.

Mapping

Now that we have created a basic lens to re-format the input table, we can use the lens in a mapping. For this, we start with the mapping template provided in the tutorial files. This template already contains basic mappings that define individuals of the classes `:Museum` and `:Worker`.

We now want to set the `:yearlyProfit` datatype property for all museums. We can achieve that by adding the following mapping:

```
mappingId    MAPID-museum-profit
target       data:museum/{museum_id} :yearlyProfit
{yearly_profit} .
source       SELECT museum_id, yearly_profit FROM
lenses.museum_projection;
```

After doing that, we can copy the `lenses.json` and `mapping.obda` files into the Ontop endpoint `input` directory, as described in the [setup page](#) and start the endpoint. Once the endpoint is started, we can open the SPARQL query editor at <http://localhost:8080> and run the following query to test it:

```
PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?profit WHERE {
    ?museum a :Museum .
    ?museum :name ?name .
    ?museum :yearlyProfit ?profit .
}
```

If the lenses and mappings were constructed correctly, this query should return a list of museum names, together with different values for their yearly profit.

NOTE

Notice how by changing the source query by adding `ratings`, `yearly_income`, or `yearly_spendings` to it, the execution of the mapping fails, stating that the columns were not found. This is because they were hidden by the lens.

Filter

Another use case of basic lenses is to filter out some specific rows from the input table. As an example, we will look at the table `workers`. This table has the following schema:

column	type
worker_id	integer
full_name	string
role	string
museum_id	integer
titles	array of strings
access_level	integer

The column `worker_id` is a primary key. The column `museum_id` is a foreign key that references the table `museums`.

The column `role` is a string that indicates the name of the worker's role. It can take the following three values: `"manager"`, `"guide"`, and `"guard"`. For our VKG, we decide that we want to designate all managers as individuals of the class `:Manager`. One way to achieve this is to use the filter feature of basic lenses.

We can prepare our `lenses.json` file like this:

```
json
{
  "relations": [
    {
      "name": ["lenses", "managers_filter"],
      "baseRelation": ["workers"],
      "filterExpression": ...,
      "type": "BasicLens"
    }
  ]
}
```

The `columns` field can be removed, as it is not required for this example. Now, we just need to define a value for the field `filterExpression` that ignores all rows for which the column `role` is not equal to `"manager"`. The `filterExpression` is defined as a SQL expression in the same style as SQL `WHERE` clauses.

After making this change, the full `lenses.json` file should look like this:

```

{
  "relations": [
    {
      "name": ["lenses", "managers_filter"],
      "baseRelation": ["workers"],
      "filterExpression": "role =
'manager'",
      "type": "BasicLens"
    }
  ]
}

```

Mapping

We can once again use the generated lens in our mapping file. For this, we will again extend the mapping template with one new mapping:

```

mappingId    MAPID-worker-managers
target       data:worker/{worker_id} a :Manager .
source       SELECT worker_id FROM
lenses.managers_filter;

```

Then, we start the Ontop endpoint and open the SPARQL editor to run the following query:

```

PREFIX : <http://example.org/museum_kg/>
SELECT ?name WHERE {
  ?worker a :Manager .
  ?worker :name ?name .
}

```

```
}
```

This should result in a list of 3 manager names.

WARNING

All examples in the lens sections are compatible with each other. However, all lenses referenced in the mapping file need to be included in `lenses.json`. If you wish to continuously extend the mapping file throughout the tutorial, you will also need to keep all previous lenses in the `relations` list of the lenses file, otherwise, Ontop will throw an error. Alternatively, you can always start each of the sections from a new mapping template file. This way, only the lenses of the current exercise have to be included.

As an extension to this exercise, you can now try to define similar lenses and mappings for the `"guide"` and `"guard"` roles, assigning them to the classes `:Guide` and `:Guard`, respectively.

Adding Constraints

Another valuable feature of lenses is adding further constraints to relations. This feature is supported for all types of lenses, but we will cover it in this section only.

Generally, Ontop can infer many constraints from the base relation used by a lens. For instance, the field `museum_id` in the table `museumS` is a primary key, so it is *unique* and *not null*. Our previously defined lens will be able to infer that the output relation is still unique and not null. On the other hand, for composite primary keys, if one part of the composite key is hidden by a lens, then Ontop knows that the remaining part is no longer guaranteed to be unique.

In many instances, however, expert knowledge can be used to define further constraints for lenses. A full list of all supported

constraints and how they can be defined can be found in the [documentation of lenses](#).

For this section, we want to use our expert knowledge of the table `workers` to provide the following constraints:

1. The name of a worker is *unique* and *not null*.
2. The role of a worker is *not null*
3. There is a functional dependency from *role* to *access_level*.
4. The column `museum_id` is a foreign key that references the table `museums` (DuckDB does not have a notion of foreign keys, so it is useful to add it explicitly).
5. The `role` column is *IRI-safe* (all possible values of the column can be safely included in an IRI without further encoding).

We can create a basic lens over the table `workers` to define these constraints, following the guidelines from the [documentation of lenses](#). A possible solution could look like this:

```

{
  "relations": [
    {
      "name": ["lenses",
"workers_constraints"],
      "baseRelation": ["workers"],
      "uniqueConstraints": {
        "added": [
          {
            "name": "uc",
            "determinants":
["full_name"]
          }
        ]
      },
      "nonNullConstraints": {
        "added": [
          "full_name",
          "role"
        ]
      },
      "otherFunctionalDependencies": {
        "added": [

```

```

        {
            "determinants": ["role"],
            "dependents":
["access_level"]
        }
    ],
    },
    "foreignKeys": {
        "added": [
            {
                "name": "fk",
                "from": ["museum_id"],
                "to": {
                    "relation":
["museumS"],
                    "columns":
["museum_id"]
                }
            }
        ]
    },
    "iriSafeConstraints": {
        "added": [
            "role"
        ]
    },
    "type": "BasicLens"
}
]
}

```

Lenses can reference each other! As a further exercise, you can try using this newly created lens as the `baseRelation` of the managers filter from before. Once that is done, you can query it again - the results should be the same as before.

NOTE

We could also have added all of these constraints to the same `managers_filter` lens from the previous section to achieve the same results. This has the advantage of reducing the work required for the lens setup and reducing the total number of relations accessible by Ontop, but it is less flexible, as we would have to copy all

the constraints to the `guides_filter` and `guard_filter` lenses as well, resulting in a lot of duplication.

Generally, the optimal solution depends on the specific scenario.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Database and Ontop Setup](#)

[Join Lens →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Join Lens

Join lenses can be used to combine multiple relations into one. Additionally, a filter expression can be provided as a join condition.

For this example, we will look at the tables `museums` and `workers` from the DuckDB database. For each individual of the `:Worker` class, we want to set its property `:workAddress` to the address of the museum they work at. The tables have the following schemas:

museums

column	type
museum_id	integer
name	string
address	string
yearly_income	integer
yearly_spendings	integer
ratings	array of floats

workers

column	type
worker_id	integer
full_name	string
role	string
museum_id	integer
titles	array of strings
access_level	integer

Notably, the table `workers` has the column `museum_id` which references the primary key of the table `museums`. In SQL, we can run a `JOIN` query over these two tables to combine all rows of `worker` with their corresponding museums. In Ontop, we can instead create a join lens, that can be referenced by the mapping.

The join lens has the following structure:

```

{
  "name": [String],
  "join": {
    "relations": [[String]],
    "columnPrefixes": [String]
  },
  "columns": {
    "added": [{
      "name": String,
      "expression": String
    }],
    "hidden": [String]
  },
  "filterExpression": String,
  "type": "JoinLens"
}

```

The `join` field takes an object consisting of a list of relation references and a list of *column prefixes*. For each relation, its corresponding column prefix will be prepended to the names of all of its columns.

The fields `columns` and `filterExpression` work exactly the way they worked for *basic lenses*, with the only reference being that now, column names have to be combined with the individual relation's prefix when referencing them in expressions.

In this example, the relations we use are `museums` and `workers`, and we choose the prefixes `m_` and `w_` for them respectively. Since we want to perform a `JOIN` operation, rather than a cross-product, we also have to supply the filter expression `m_museum_id = w_museum_id`. After including these values, the `lenses.json` file should look like this:

```

{
  "relations": [
    {
      "name": ["lenses", "museums_workers"],
      "join": {
        "relations": [
          ["museums"],
          ["workers"]
        ],
        "columnPrefixes": [
          "m_",
          "w_"
        ]
      },
      "columns": ...,
      "filterExpression": "m_museum_id =
w_museum_id",
      "type": "JoinLens"
    }
  ]
}

```

The only remaining field is `columns`. As mentioned earlier, this field is handled analogously to the basic lens `columns` field, allowing the user to add and remove specific columns. For this example, we want to set the property `:workAddress` for each `:Worker` individual. Because of that, we only require the column `worker_id` from `workers` and `address` from `museums` - all other columns can be hidden, and no column has to be added:

```

{
  "relations": [
    {
      "name": ["lenses", "museum_workers"],
      "join": {
        "relations": [
          ["museums"],
          ["workers"]
        ],
        "columnPrefixes": [
          "m_",
          "w_"
        ]
      }
    }
  ]
}

```

```

    },
    "columns": {
      "added": [],
      "hidden": [
        "m_museum_id",
        "m_name",
        "m_yearly_income",
        "m_yearly_spendings",
        "m_ratings",
        "w_full_name",
        "w_role",
        "w_museum_id",
        "w_titles",
        "w_access_level"
      ]
    },
    "filterExpression": "m_museum_id =
w_museum_id",
    "type": "JoinLens"
  }
]
}

```

Mapping

Now that the lens file is created, we can construct our mapping. For this, we once again start from the mapping template file provided with the tutorial files.

We can add a mapping entry to it, referencing the newly created lens.

```

mappingId    MAPID-museum-worker-address
target       data:worker/{w_worker_id} :workAddress
              {m_address} .
source       SELECT w_worker_id, m_address FROM
              lenses.museum_workers;

```

Now, we can test the lens and mapping, by copying the corresponding files to the Ontop endpoint directory and running the following query:

```
PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?address WHERE {
    ?worker a :Worker .
    ?worker :name ?name .
    ?worker :workAddress ?address .
}
```

If everything was prepared correctly, this should result in a list of employee names, together with the address of the museum they work at.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Basic Lens](#)

[Union Lens →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

SQL Lens

SQL lenses are a special type of lens that can generate virtual views over any number of base relations through an arbitrary SQL query. While this allows for more flexibility, as any SQL functionalities can be used by this lens, it risks to obfuscate the inner workings towards Ontop as it may not be handled by the Ontop's SQL parser, preventing it from performing any meaningful inference and optimization.

One major use case for SQL lenses is to perform complex operations that are currently not supported by other Ontop lenses. For this example, we will look at the table `art_exhibits`. This table has the following schema:

column	type
exhibit_id	integer
name	string
artist_name	string
museum_id	integer

We now want to gather more information on artists. Including how many exhibits they created and in how many museums their works appear. In SQL, this can be achieved easily by running aggregate functions on a `GROUP BY` query, but there is no corresponding lens for aggregate functions in Ontop. We, therefore, have to take advantage of the SQL lens.

This lens has the following structure:

```
{
  "name": [String],
  "query": String,
  "type": "SQLLens"
}
```

json

```
}
}
```

Here, the `query` field is a single SQL query that projects all attributes that are of interest to us. In our example, we want to group all artists' names and count their number of exhibits and distinct museums. A corresponding SQL lens could look like this:

```
{
  "name": ["lenses", "artists"],
  "query": "SELECT artist_name,
COUNT(exhibit_id) as exhibits, COUNT(DISTINCT
museum_id) as museums FROM art_exhibits GROUP BY
artist_name",
  "type": "SQLLens"
}
```

This will now create a virtual relation inside Ontop, that has the columns `artist_name`, `exhibits`, and `museums`.

Mapping

Finally, we just need to create a mapping entry for the artist, extending our mapping template:

```
mappingId    MAPID-artists
target       data:artist/{artist_name} a :Artist ;
:name {artist_name}^^xsd:string ; :exhibitCount
{exhibits}^^xsd:integer ; :museumCount
{museums}^^xsd:integer.
source       SELECT artist_name, exhibits, museums
FROM lenses.artists;
```

NOTE

In the `target` clause of the mapping, our datatype properties have to be marked by their individual types. This is because Ontop can no longer infer the types of the columns we are using, as they are obfuscated by the SQL lens.

To test our lenses and mapping, let us run the Ontop endpoint, copying the `lenses.json` and `mapping.obda` files into the endpoint's directory. Then, we can run the following SPARQL query:

```
PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?exhibits ?museums WHERE {
  ?artist a :Artist .
  ?artist :name ?name .
  ?artist :exhibitCount ?exhibits .
  ?artist :museumCount ?museums .
}
```

If everything was done correctly, you should get a list of three artists with the number of their exhibits and the number of museums they are featured in.

Adding Unique Constraints

Ontop cannot infer unique constraints from the expressions used in SQL lenses. However, as the user, it is clear to us that the field `artist_name` will be *unique*, as it is used by the `GROUP BY` clause. In such cases, explicitly adding unique constraints is a useful feature. Similarly to how it was shown in the [basic lens section](#), we can achieve this by adding an additional field to the lens:

```
{
  "relations":
  [
    {
      "name": ["lenses", "artists"],
      "query": "SELECT artist_name,
COUNT(exhibit_id) as exhibits, COUNT(DISTINCT
museum_id) as museums FROM art_exhibits GROUP BY
artist_name",
      "type": "SQLLens",
      "uniqueConstraints": {
        "added": [
          {
            "name": "uc",
```

```

    "determinants":
    [
      [
        "artist_name"
      ]
    ]
  }
}

```

While this will not change the output of the sample query, it may help Ontop optimize its queries in specific instances (see [primary key](#)).

WARNING

It is advised to be cautious when using SQL lenses. Generally, they should not refer to other lenses if the SQL expression is complex, and they may not be able to infer integrity constraints. For more information, please visit the [documentation page of lenses](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Flatten Lens](#)
[External tutorials →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Union Lens

Union lenses will be supported starting with version 5.1.0 (beta).

Union lenses can be used to concatenate multiple relations with the same schema into one. Additionally, a *provenance* field can be added to each row, determining the source relation the row originates from.

For this example, we will look at the tables `nature_exhibits`, `historical_exhibits` and `art_exhibits` from the DuckDB database. We want to combine them into a single relation containing all exhibits.

The tables `nature_exhibits` and `historical_exhibits` have the following schema:

column	type
exhibit_id	integer
name	string
museum_id	integer

The table `art_exhibits` has the following schema:

column	type
exhibit_id	integer
name	string
artist_name	string
museum_id	integer

For each table, `exhibit_id` is a primary key and `museum_id` is a foreign key, referencing the table `museums`.

Relations with the same schema

This concatenation can be achieved using a union lens. However, as all base relations of a union lens must have the exact same columns, we will only be working on the tables

`nature_exhibits` and `historical_exhibits` at first.

The union lens has the following structure:

```

{
  "name": [String],
  "unionRelations": [[String]],
  "makeDistinct": boolean,
  "provenanceColumn": String,
  "type": "UnionLens"
}

```

Here, `unionRelations` is a list of relations that should be concatenated, `makeDistinct` indicates if a distinct constraint should be enforced on the result, and `provenanceColumn` is an optional parameter, indicating the name of the provenance column in the result, which tells us what relation each row originated from.

As the tables `nature_exhibits` and `historical_exhibits` have the same set of columns, we can reference them from a union lens to create a new, concatenated view.

WARNING

The columns of all tables used by a union lens must be *exactly* equal. Their columns must have *the same names* and the exact *same data types*. The order of the columns does not matter.

```

{
  "relations": [
    {
      "name": ["lenses", "all_exhibits"],
      "unionRelations": [

```

```

        ["historical_exhibits"],
        ["nature_exhibits"]
    ],
    "provenanceColumn": ...,
    "makeDistinct": ...,
    "type": "UnionLens"
}
]
}

```

The remaining fields are `provenanceColumn` and `makeDistinct`. As we do not expect the entries of any of the tables to be equal, we can just neglect the `makeDistinct` parameter. To preserve the source of each entry, we want to include a provenance column. We can call it `exhibit_type`. This results in the following `lenses.json` file:

```

{
  "relations": [
    {
      "name": ["lenses", "all_exhibits"],
      "unionRelations": [
        ["historical_exhibits"],
        ["nature_exhibits"]
      ],
      "provenanceColumn": "exhibit_type",
      "type": "UnionLens"
    }
  ]
}

```

NOTE

If the `provenanceColumn` field is not provided, then no information on the source relation will be preserved for the concatenated rows. If the field `makeDistinct` is not provided, its default value is assumed to be `false`.

NOTE

Including a `provenanceColumn` in a UnionLens where

each of the source relations is distinct will render the `makeDistinct` field redundant. This is because, under a union, the composite uniqueness constraint `(provenanceColumns, childUniquenessConstraint)` will always hold. On the other hand, if no provenance column is included, then uniqueness constraints from the base relations will be lost, as there is no guarantee that a specific value does not appear again in a different table.

Mapping

Now, let us use this lens definition in our mapping. Starting from the mapping template file, add the following mapping entry:

```
mappingId    MAPID-exhibits
target       data:exhibit/{exhibit_id} a :Exhibit ;
:name {name} ; :displayedIn
data:museum/{museum_id} ; :exhibitType
{exhibit_type} .
source       SELECT exhibit_id, name, museum_id,
exhibit_type FROM lenses.all_exhibits;
```

Starting the Ontop SPARQL endpoint using this mapping and lens file, we can now run the SPARQL query:

```
PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?type WHERE {
  ?exhibit a :Exhibit .
  ?exhibit :name ?name .
  ?exhibit :exhibitType ?type .
}
```

This should result in a set of exhibit names that have the values `historical_exhibits` and `nature_exhibits` as their exhibit type.

Relations with different schemas

If we further want to include the table `art_exhibits` to the union lens, we cannot just add it to the list of `unionRelations`. This is because it has one additional column, `artist_name`, that does not appear in the other tables. To work around this issue, we can take advantage of basic lenses, putting them "underneath" the union lens, to ensure the equality of columns.

There are two possibilities to achieve this:

1. Hide the conflicting column(s) from its/their table(s)
2. Add the conflicting column(s) to the tables that do not include them.

To retain a maximum of information, we will choose the second approach for this example. That means, that we have to construct basic lenses to add the column `artist_name` to the tables `historical_exhibits` and `nature_exhibits`. One possible way to achieve this is by adding new columns that have `NULL` as their expressions.

```

{
  "relations": [
    {
      "name": ["lenses",
"historical_exhibits_extended"],
      "baseRelation":
["historical_exhibits"],
      "columns": {
        "added": [
          {
            "name": "artist_name",
            "expression": "'None'"
          }
        ],
        "hidden": []
      },
      "type": "BasicLens"
    }
  ]
}

```

After doing the same for the `nature_exhibits` table, referencing the new extended lenses from the union lens, and

adding `art_exhibits` as one of its union relations, get the following lens file:

```

{
  "relations": [
    {
      "name": ["lenses",
"historical_exhibits_extended"],
      "baseRelation":
["historical_exhibits"],
      "columns": {
        "added": [
          {
            "name": "artist_name",
            "expression": "'None'"
          }
        ],
        "hidden": []
      },
      "type": "BasicLens"
    },
    {
      "name": ["lenses",
"nature_exhibits_extended"],
      "baseRelation": ["nature_exhibits"],
      "columns": {
        "added": [
          {
            "name": "artist_name",
            "expression": "'None'"
          }
        ],
        "hidden": []
      },
      "type": "BasicLens"
    },
    {
      "name": ["lenses", "all_exhibits"],
      "unionRelations": [
        ["lenses",
"historical_exhibits_extended"],
        ["lenses",
"nature_exhibits_extended"],
        ["art_exhibits"]
      ],
      "provenanceColumn": "exhibit_type",

```

```

        "type": "UnionLens"
    }
]
}

```

Mapping

We can now extend the mapping file to also include artist names:

```

mappingId    MAPID-exhibits
target       data:exhibit/{exhibit_id} a :Exhibit ;
:name {name} ; :displayedIn
data:museum/{museum_id} ; :exhibitType
{exhibit_type} ; :artistName {artist_name} .
source       SELECT exhibit_id, name, museum_id,
exhibit_type, artist_name FROM
lenses.all_exhibits;

```

Now, running this slightly modified SPARQL query:

```

PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?type ?artist WHERE {
    ?exhibit a :Exhibit .
    ?exhibit :name ?name .
    ?exhibit :exhibitType ?type .
    ?exhibit :artistName ?artist
}

```

we will once again get all earlier results, in addition to all exhibits contained in the `art_exhibit` table. While the earlier results will have the value `None` as their artist name, the `art_exhibits` entries will include the name of their artists.

As a further exercise, notice how the values of `:exhibitType` are rather ugly: `"art_exhibits"`, `"lenses.historical_exhibits_extended"`, and `"lenses.nature_exhibits_extended"`. Try adding a new basic lens over the union lens that transforms these into the values `"art"`, `"historical"`, and `"nature"` instead.

Hint: Look at the SQL function `REPLACE`. Could it be used in the `expression` field of an added column to get rid of the `_exhibits`, `_extended`, and `lenses.` part?

Notice how the union lens allowed you to perform this operation on all three of its base relations by just defining it once. Without the union lens, you would have required three such basic lenses, one for each table, with the exact same contents.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Join Lens](#)

[Flatten Lens →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Flatten Lens

Flatten lenses are supported since 5.1.0.

Flattening or unnesting an array is the process of transforming a nested array into an array of lower dimensionality, by "pulling" each nested entry into its "outer" entry. In databases, specifically, it represents a function that takes a column containing an array and transforms it into a table that has each of the *outer-most* elements as one of its rows.

Flattening is need when we want to map values inside arrays.

In Ontop, a flatten lens is a type of lens over a single base relation that takes as an input the column that should be flattened, the names of columns that should be retained after flattening, and the name of the column that should be added containing the flattened output. Additionally, the name for a *position* column can be passed to the lens. When flattening, the *position* column will hold the index of the current row's flattened element in the original array.

For this example, we will first look at the table `workers`. This table has the following schema:

column	type
worker_id	integer
full_name	string
role	string
museum_id	integer
titles	array of strings
access_level	integer

The column `titles` is an array of strings that contains all the

degrees and titles the employee has received. Our goal is to populate the datatype properties `:hasTitle` defined from the `:Worker` class to any number of string literals taken from the table, and `:preferredTitle`, defined from the `:Worker` class to a single literal, which is the first entry of the `titles` array. To do so efficiently, we need to flatten the `titles` column.

The flatten lens has the following structure:

```

{
  "name": [String],
  "baseRelation": [String]
  "flattenedColumn": {
    "name": String,
    "datatype": String
  },
  "columns": {
    "kept": [String],
    "new": String,
    "position": String
  },
  "type": "FlattenLens"
}

```

The field `kept` takes a list of column names from the original table that should be retained after flattening. The field `new` takes the name of the column that contains the flattened output and the field `position` takes the name of the column that contains the index of the current output. The flattened column takes two arguments: `name` determines the name of the column that should be flattened, and `datatype` indicates the column's type. Depending on the SQL dialect, this can vary from `ARRAY<T>` or `T[]` where `T` is a different SQL data type to `JSON` or `VARCHAR` if arrays are represented and flattened in JSON format.

In DuckDB, the array data type is defined as `T[]`, so in our specific case, the field `datatype` will take the value `STRING[]`.

A possible `lenses.json` file for this task may look like this:

```


```

```
{
  "relations": [
    {
      "name": ["lenses",
"flattened_titles"],
      "baseRelation": ["workers"],
      "flattenedColumn": {
        "name": "titles",
        "datatype": "STRING[]"
      },
      "columns": {
        "kept": [
          "worker_id"
        ],
        "new": "title",
        "position": "index"
      }
    }
  ]
}
```

As we just need to assign a title to a `:Worker` individual, the only column that has to be kept is `worker_id`.

Mapping

Now, we can already define the mapping entries for the wanted properties:

```
mappingId  MAPID-has-title
target     data:worker/{worker_id} :hasTitle
{title} .
source     SELECT worker_id, title FROM
lenses.flattened_titles;

mappingId  MAPID-preferred-title
target     data:worker/{worker_id}
:preferredTitle {title} .
source     SELECT worker_id, title FROM
lenses.flattened_titles WHERE index = 1;
```

NOTE

Instead of using a source query with a `WHERE` condition in the second mapping, we could also wrap a basic lens around the flatten lens that performs the filter operation, but we kept it like this for the sake of simplicity.

Let us now run a SPARQL query over the Ontop endpoint to test our mappings and lenses.

```
PREFIX : <http://example.org/museum_kg/>
SELECT ?name ?title ?prefTitle WHERE {
  ?worker a :Worker .
  ?worker :name ?name .
  ?worker :hasTitle ?title .
  ?worker :preferredTitle ?prefTitle
}
```

This should result in a list of all employees, together with their titles and their preferred title.

Flattening other types of arrays

The flatten lens works on all types of arrays, and, depending on the capabilities of the dialect, can infer the output type of the flattened column. As a further exercise, try looking at the table

`museums`.

column	type
museum_id	integer
name	string
address	string
yearly_income	integer
yearly_spendings	integer
ratings	array of floats

The field `ratings` is an array of floating point numbers between 1 and 10. Try creating a flatten lens that can unnest this array to populate the datatype property `:hasRating`! Ontop

will be able to automatically detect that the flattened column has the type `FLOAT`. Keep in mind that we do not necessarily need a "position" column for this use case.

WARNING

When arrays in a given dialect are defined as `ARRAY<T>`, `T[]`, or similarly, Ontop is able to infer the data type of the output column after flattening. However, when this is not the case (either the array was provided as JSON or the array data type of the dialect is simply called `ARRAY` or similarly), Ontop cannot perform this inference. In those cases, it is suggested to put a basic lens over the flatten lens that explicitly performs a `CAST` on the output, to allow Ontop to know the column type once again.

WARNING

The level of support for the flatten lens depends strongly on the dialect. Please consult the [flatten lens documentation page](#) for more info on each supported dialect.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Union Lens](#)

[SQL Lens →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

External tutorials

- [Destination tutorial](#) : about lodging businesses, municipalities and weather observations, using standard ontologies (schema.org, GeoSPARQL and Semantic Sensor Network).
- [A tutorial using the Bgee database](#) : Code for the paper [Accessing scientific data through knowledge graphs with Ontop](#)

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← [SQL Lens](#)

Community

- People
- Organizations
 - Free University of Bozen-Bolzano
 - Ontopic s.r.l.
 - Birkbeck, University of London
- Support

Contributing

- How to contribute
- Bug report
- Pull request
- Documentation

Organizations

Here we mention some of the organizations actively involved into the development of Ontop.

Free University of Bozen-Bolzano

The [KRDB research group](#) initiated and continues to lead the development of Ontop.

Ontopic s.r.l.

[Ontopic](#) has been co-founded by members of the research group at the Free University of Bozen-Bolzano that started Ontop. Ontopic provides a complete [suite](#) of solutions for leveraging the potential of Ontop and Virtual Knowledge Graphs.

Birkbeck, University of London

The [Birkbeck](#) group have been extensively involved in the development of Ontop since 2012. They have worked on the theoretical foundations of the VKG approach, including query rewriting and optimisation, and contributed to the code base.

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Community

- People
- Organizations
- Support
 - FAQ
 - Mailing list (Google group)
 - Bug tracking system
 - Commercial support

Contributing

- How to contribute
- Bug report
- Pull request
- Documentation

Support

FAQ

Please visit our pages about [Frequently Asked Questions](#).

Mailing list (Google group)

We have a [mailing list \(Google group\)](#) for anything related to support, discussion and announcements. Please join for any inquiry.

Bug tracking system

To see the list of open issues/bugs, visit [our GitHub tracker](#)

Commercial support

[Ontopic](#) is offering commercial support for Ontop.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Community

- People
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Reporting a possible bug
- Asking for help or information
- Pull request
- Documentation

Bug report

If you are sure you found a bug, use [GitHub Issues](#) to submit the report; otherwise, start a discussion in the [mailing list](#) .

When reporting a bug, please make sure that you are using [the latest version of Ontop](#) and provide a minimal reproducible example.

Reporting a possible bug

- Perform a search between current and past issues. If it is an open issue, add a comment to the existing issue instead of opening a new one.
- Let us know which version and which setting of Ontop you are using (Protégé, SPARQL endpoint, CLI, Java API)
- State what happened and also state what you expected to see.
- If you have precise steps to reproduce the bug, explain each step and send us a set of ontology/mappings/query (as small as possible) so we can reproduce the problem. Otherwise send us the complete ontology, mapping and query files.
- If you cannot reliably reproduce the test provide details about how often the problem happens and under which conditions it normally happens.
- If the program generated an error, report what the error message was and how it occurred.
- Add the log file in debug mode to help us to track the issue (more information follows).

How to generate the log file

Enable Protégé log

You should go to the folder *conf* of Protégé and modify the line of the *logback.xml* file from `<root Level="info">` to `<root`

`Level="debug">` . When you restart Protégé you will have more information in the command line, that can be useful to understand the problem. You can send us the generated log file from Protégé starting from *Window -> Show log... -> Show log file -> protege.log*

Enable Command Line Interface log

Go to the folder *log* and modify the line of the *logback.xml* file from `<root Level="ERROR">` to `<root Level="debug">` .

Enable logging in the Docker image

Set the environment variable `ONTOP_LOG_LEVEL` to `debug` (`ONTOP_LOG_LEVEL=debug`).

Asking for help or information

See [the support page](#).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← How to contribute](#)

[Pull request →](#)

Community

- People
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Pull request
- Contribution License Agreement (for core developers)
- Documentation

Pull request

Contributions to the project are welcome and encouraged. You can propose some Pull Requests (PR) on Github. By making a PR on the Github platform, you declare to agree with the standard "*inbound = outbound*" licensing practice, as foreseen by the [Github terms](#) . There is no need anymore for regular contributors to sign a CLA.

If you are thinking about implementing a new feature, feel free to engage a discussion with the team through the mailing list or through another medium.

Contribution License Agreement (for core developers)

Core developers should sign a Contribution License Agreement (CLA) that allows FUB and the Ontop team to redistribute their work under the terms of the Apache 2.0 license. If the contributor works for an institution that owns the work being contributed by the contributor, then a corporate CLA must be additionally signed. The CLAs are available for download below, once signed please email a scanned copy of them.

- Individual Contributor License Agreement (CLA)
- Corporate License Agreement

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Community

[People](#)

[Organizations](#)

[Support](#)

Contributing

[How to contribute](#)

[Bug report](#)

[Pull request](#)

[Documentation](#)

[Small improvements](#)

[Testing the documentation locally](#)

Documentation

The source code of the documentation is on [Github](#) and is powered by [VuePress](#).

Once a commit is pushed to the Github repository, this public documentation is automatically rebuilt by Travis CI.

Small improvements

Have you found a typo, a broken URL or another small problem? Please propose a fix immediately (before forgetting it), by clicking on the link [Edit this page](#) at the bottom of the page.

Testing the documentation locally

For a more involved work on the documentation, we recommend you to test your changes on your local machine before pushing them to Github.

```
git clone git@github.com:ontop/ontop-website.git
cd ontop-website

# Install the dependencies (e.g. vuepress)
yarn install

# start the local server
yarn docs:dev

# commit your changes and push them back to Github
```

[VuePress](#) requires Node.js >= 8.6.

[Edit this page](#)

[← Pull request](#)

Research

Research team

Publications

Journals

Conferences

Workshops

Tutorials and Demos

Master theses

Theory

Notation

Formal characterization of IQs

Intermediate query optimization

Publications

Journals

1. Virtual Knowledge Graphs: An Overview of Systems and Use Cases. Guohui Xiao, Linfang Ding, Benjamin Cogrel, Diego Calvanese. Data Intelligence. 2019.
2. Ontop-spatial: Ontop of Geospatial Databases. Konstantina Bereta, Guohui Xiao, and Manolis Koubarakis. Journal of Web Semantics. 2019.
3. Ontology-Based Access To Temporal Data With Ontop: A Framework Proposal. Elem Güzel Kalaycı, Sebastian Brandt, Diego Calvanese, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyashev. International Journal of Applied Mathematics and Computer Science. 2019.
4. VIG: Data Scaling for OBDA Benchmarks. Davide Lanti, Guohui Xiao, and Diego Calvanese. Semantic Web Journal. 2019.
5. Ontology-based Data Access - Beyond relational sources. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, Guohui Xiao. Intelligenza Artificiale. 2019.
6. Querying Log Data with Metric Temporal Logic. Sebastian Brandt, Elem Güzel Kalaycı, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyashev. Journal of Artificial Intelligence Research, 2018.
7. Ontop: Answering SPARQL Queries over Relational Databases. Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Semantic Web Journal. 2017 (SWJ 2016 Outstanding Paper Award)
8. Ontology Based Data Access in Statoil. Evgeny Kharlamov, Dag Hovland, Martin G. Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis E. Ioannidis, Yannis Kotidis, Manolis Koubarakis, Arild Waaler.

- Journal of Web Semantics, 2017.
9. Efficient SPARQL-to-SQL with R2RML mappings. Mariano Rodriguez-Muro and Martin Rezk, J. of Web Semantics. 2015
10. Optique --- zooming in on big data access. Martin Giese, Peter Haase, Ernesto Jiménez-Ruiz, Davide Lanti, Özgür Özçep, Martin Rezk, Riccardo Rosati, Ahmet Soylu, Guillermo Vega-Gorgojo, Arild Waaler, and Guohui Xiao. IEEE Computer. 2015

Conferences

1. The Virtual Knowledge Graph System Ontop. Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalayci, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. ISWC (Resource), 2020.
2. Semantic Integration of Bosch Manufacturing Data Using Virtual Knowledge Graphs. Elem Güzel Kalaycı, Irlan Grangel González, Felix Lösch, Guohui Xiao, Anees ul-Mehdi, Evgeny Kharlamov, and Diego Calvanese. ISWC (In-Use), 2020.
3. Enriching Ontology-based Data Access with Provenance. Diego Calvanese, Davide Lanti, Ana Ozaki, Rafael Penaloza, and Guohui Xiao. IJCAI, 2019.
4. Efficient Handling of SPARQL OPTIONAL for OBDA. Guohui Xiao, Roman Kontchakov, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. ISWC (Research), 2018.
5. Ontology-based data access: A survey. Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, Michael Zakharyashev. IJCAI. 2018.
6. Ontop-temporal: A Tool for Ontology-based Query Answering over Temporal Data. Elem Güzel Kalayci, Guohui Xiao, Vladislav Ryzhikov, Tahir Emre Kalayci, and Diego Calvanese. CIKM. 2018.
7. Efficient Ontology-Based Data Integration with canonical IRIs. Guohui Xiao, Dag Hovland, Dimitris Bilidas, Martin Rezk, Martin Giese, and Diego Calvanese. ESWC, 2018.
8. Cost-Driven Ontology-Based Data Access. Davide Lanti, Guohui Xiao, Diego Calvanese. ISWC (Research), 2017.
9. Semantic Rule-Based Equipment Diagnostics. Gulnar Mehdi,

- Evgeny Kharlamov, Ognjen Savkovic, Guohui Xiao, Elem Güzel Kalayci, Sebastian Brandt, Ian Horrocks, Mikhail Roshchin, Thomas A. Runkler. ISWC (In-Use), 2017. ([Best In-Use Paper Award](#))
10. Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic. Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, Michael Zakharyashev. AAAI, 2017.
11. Beyond OWL 2 QL in OBDA: Rewritings and Approximations. Elena Botoeva, Diego Calvanese, Valerio Santarelli, Domenico Fabio Savo, Alessandro Solimando and Guohui Xiao. AAAI, 2016.
12. Ontology-Based Data Access for Maritime Security. Stefan Brüggemann, Konstantina Bereta, Guohui Xiao and Manolis Koubarakis. ESWC (In-Use), 2016.
13. OBDA Constraints for Effective Query Answering. Dag Hovland, Davide Lanti, Martin Rezk and Guohui Xiao. RuleML, 2016.
14. Towards Practical OBDA with Temporal Ontologies (Position Paper). Diego Calvanese, Elem Güzel Kalayci, Vladislav Ryzhikov and Guohui Xiao. RR, 2016.
15. Ontology-based Integration of Cross-linked Datasets. Diego Calvanese, Martin Giese, Dag Hovland and Martin Rezk (ISWC 2015), 2015.
16. The NPD benchmark: Reality check for OBDA systems. Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. In Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT 2015). ACM Press, 2015.
17. Answering SPARQL Queries over Databases and under OWL 2 QL Entailment Regime. Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyashev. International Semantic Web Conference (ISWC), 2014.
18. Rules and Ontology Based Data Access. Guohui Xiao, Martin Rezk, Mariano Rodriguez-Muro, and Diego Calvanese. Eighth International Conference on Web Reasoning and Rule Systems, (RR), 2014.
19. High Performance Query Answering over DL-Lite Ontologies.

- Mariano Rodriguez-Muro, Diego Calvanese. KR 2012
20. Quest: Efficient SPARQL-to-SQL for RDF and OWL. Mariano Rodriguez-Muro, Josef Hardi, Diego Calvanese. ISWC (Posters & Demos) 2012

Workshops

1. Fast and Simple Data Scaling for OBDA Benchmarks. Davide Lanti, Guohui Xiao and Diego Calvanese. BLINK@ISWC, 2016.
2. OBDA Beyond Relational DBs: A Study for MongoDB. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk and Guohui Xiao. Description Logics, 2016.
3. The ontop framework for ontology based data access. Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, Mindaugas Slusnys, and Guohui Xiao. CSWS (Posters and Demos), Wuhan, China, August 2014.
4. The NPD benchmark for OBDA systems. Davide Lanti, Martin Rezk, Mindaugas Slusnys, Guohui Xiao, and Diego Calvanese. SSWS 2014
5. A scalable benchmark for OBDA systems: Preliminary report. Diego Calvanese, Davide Lanti, Martin Rezk, Mindaugas Slusnys, and Guohui Xiao. ORE 2014
6. Query Rewriting and Optimisation with Database Dependencies in Ontop. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev: Description Logics 2013
7. Evaluating SPARQL-to-SQL Translation in Ontop. Mariano Rodriguez-Muro, Martín Rezk, Josef Hardi, Mindaugas Slusnys, Timea Bagosi, Diego Calvanese: ORE 2013
8. OBDA with Ontop. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev: ORE 2013
9. Ontop at Work. Mariano Rodriguez-Muro, Roman Kontchakov, Michael Zakharyashev. OWLED 2013
10. On Rewriting, Answering Queries in OBDA Systems for Big Data (Short Paper). Diego Calvanese, Ian Horrocks, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Michael Meier, Mariano Rodriguez-Muro, Dmitriy Zheleznyakov. OWLED 2013
11. Quest, an OWL 2 QL Reasoner for Ontology-based Data

Access. Mariano Rodriguez-Muro, Diego Calvanese. OWLED 2012

Tutorials and Demos

1. [Destination Tutorial](#) given at the Knowledge Graph Conference 2021 by Diego Calvanese, Benjamin Cogrel and Guohui Xiao.
2. [Novel Developments in Ontology-Based Data Access and Integration \(NOBDI\)](#) Diego Calvanese, Benjamin Cogrel and Guohui Xiao. Full day tutorial, 17th International Conference of the Italian Association for Artificial Intelligence -- Trento, November 20-23, 2018.
3. [Semantic Technologies for Data Access and Integration](#). Diego Calvanese and Guohui Xiao. Full day tutorial, CIKM. Torino, 26 October 2018.
4. [Ontology-based Data Access: Theory and Practice](#) . Guohui Xiao and Roman Konchakov. Full day tutorial, IJCAI 2018 Tutorial. Stockholm, July 14, 2018.
5. [Mapping Management and Expressive Ontologies in Ontology-Based Data Access](#) . Diego Calvanese, Benjamin Cogrel and Guohui Xiao. Full day tutorial, EKAW. Bologna, November 19, 2016.
6. [Ontology-Based Data Access with Ontop](#) . Benjamin Cogrel. Lecture at the ESSLLI summer school, 2016.
7. [Ontop: Answering SPARQL Queries over Relational Databases](#) . Guohui Xiao. Stanford, 2016.
8. [Ontology-based Data Access: From Theory to Practice](#) (Diego Calvanese, Martin Rezk, Guohui Xiao. ISWC 2015 Tutorial)
9. [Ontology-based data access --- Theory and Practice](#) . (Guohui Xiao, Summer School at CSWS2014, Wuhan, China)
10. [An Introduction to Description Logics and Query Rewriting](#). Roman Kontchakov, Michael Zakharyashev. Reasoning Web 2014
11. [Ontology-Based Data Access with Databases: A Short Course](#). (Roman Kontchakov, Mariano Rodriguez-Muro, Michael Zakharyashev. Reasoning Web 2013)
12. [Ontop: A Tutorial](#) (Mariano Rodriguez-Muro, Vienna'13)

Master theses

- Lukas Sundqvist: [Extending VKG Systems with RDF-star Support](#) , Free University of Bozen-Bolzano, March 2022.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Research team](#)

[Notation →](#)

Research

- Research team
- Publications

Theory

- Notation
- Formal characterization of IQs
- Intermediate query optimization

Notation

WARNING

TODO: CHECK/UPDATE (2 years old and formatted in AsciiDocs)

We assume a countably infinite domain D being queried, a finite set P of predicates disjoint from D , and an interpretation function $\llbracket \cdot \rrbracket$ defined over P , which to each predicate p in P associates a subset $\llbracket p \rrbracket$ of $(D \cup \{ \text{NULL} \})^i$, for some positive integer i . The *signature* $\text{sig}(Q)$ of a (sub)query Q is a subset of P . An the interpretation $\llbracket Q \rrbracket$ of Q is built inductively as usual, out of all $\llbracket p \rrbracket$ such that p is $\text{sig}(Q)$, by application of RA operators.

In addition, we assume a countably infinite set A of attribute names, distinct from D and P . If $\llbracket Q \rrbracket$ is the interpretation of (sub)query Q , then the function $\text{att}(\llbracket Q \rrbracket)$ associates to $\llbracket Q \rrbracket$ a finite tuple ofn distinct elements of A , and n is called the *arity* of $\llbracket Q \rrbracket$.

NOTE: It is also assumed in what follows that the algebraic expression corresponding to an intermediate query is syntactically valid, according to standard requirements on RA operators. For instance, when a projection is performed, the projected attributes must be a subset of the attributes of the argument relation. Similarly, the arguments of a union operator must have identical set of attributes (this is not natively the case of the UNION SPARQL operator, but can be enforced with a straightforward normalization).

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Research

Research team

Publications

Theory

Notation

Formal characterization of IQs

Notation

Leaf node types

Non-leaf node types

Intermediate query optimization

Formal characterization of IQs

WARNING

TODO: CHECK/UPDATE (2 years old and formatted in AsciiDocs)

This section aims at providing a more precise characterization of an *intermediate query* (IQ in what follows), which is the main query representation format used internally in Ontop.

An IQ is a standard (rooted and ordered) tree representation of an algebraic query expression. More exactly, for each node N of a well-formed IQ Q, the maximal subtree of Q rooted in N represents a valid algebraic expression.

In order to give a more precise characterization of this algebra, we will rely on its similarity with Relational Algebra (RA in what follows), more exactly the select/project/join/rename/union RA with named attributes (and no duplicate attribute), extended with additional operators supported by SQL DBMS (and relevant for Ontop's setting). These additional operators are:

- left outer join (which corresponds to the SPARQL OPTIONAL operator), and
- extended projection

NOTE

As of now, difference is not supported.

For readability, some similarities with the SPARQL language will also be (informally) highlighted.

NOTE: This is only one possible way of characterizing the

meaning of an IQ. Other options would be for instance relying on (a customized version of) the SPARQL algebra, or defining from scratch a full-fledged semantic.

link:notation[The first section] introduces some notational conventions, whereas the following ones present the different types of nodes which may appear in an IQ, and how they relate to RA operands and operators. In order to simplify notation, their interpretation is first given in terms of set semantics (which is also the standard interpretation of RA).

The extension to bag semantics has specific implications in an OBDA setting, and is discussed in link:bagSemantics[a dedicated Section].

Notation

Interpretation of algebraic expressions

We assume the following mutually disjoint sets:

- a countably infinite domain D being queried,
- a countably infinite set V of variables names (similar to SPARQL variables, or to a lesser extent to RA attribute names).
- a finite set P of predicates, and
- For each $p \in P$ with arity $n > 0$, a set $\{p_1, \dots, p_n\}$ of base attribute names.

A database instance will be viewed as a total function I over P , which, to each element p of its domain, associates a set $I(p)$ of partial functions from $\{p_1, \dots, p_n\}$ to D .

The *signature* $\text{sig}(M)$ of an algebraic query expression M is the set of its base operands, and it is assumed that for any well-formed M , $\text{sig}(M) \subseteq P$ holds.

Given a database instance I over P , the *interpretation* $\|M\|$ of a well-formed algebraic expression M is built inductively out of all $I(p)$ such that $p \in \text{sig}(M)$, by application of algebraic operators.

To each element of its domain, $\llbracket \cdot \rrbracket$ associates a (possibly empty) set of partial functions from a unique subset $\text{var}(M)$ of V to D . $\text{var}(M)$ will be called the *projected variables* of M , and does not depend on $\llbracket \cdot \rrbracket$, but only on M .

Interpretation of IQ nodes

If Q is a well-formed IQ, then $\text{nodes}(Q)$ will designate its nodes. And by extension, if $N \in \text{nodes}(Q)$, then $\text{sig}(N)$ (resp. $\llbracket N \rrbracket$ and $\text{var}(N)$) will designate the signature (resp. the interpretation and the projected variables) of the algebraic expression represented by (the maximal subtree of Q rooted in) N .

Boolean expressions

A *boolean expression* in what follows designates a quantifier-free first-order formula, with variables in V , constants in D and predicates among the ones listed [link](#): [in the dedicated section].

For instance, $x \neq y$, $(x \geq 2) \vee (x = z)$ and $\text{IsNotNull}(x)$ are valid expressions.

Given a boolean expression φ and a partial function $f : V \mapsto D$, the formula $\varphi[f]$ is obtained by replacing each variable x appearing in φ by $f(x)$ if defined, and by NULL otherwise. Then the valuation $\text{val}(\varphi[f])$ of $\varphi[f]$ corresponds to the one expected in SQL, and operators like "=" or $\geq$ are polymorphic. For instance, for any $d \in D$, $\text{val}(d \leq d) = \text{TRUE}$, $\text{val}(d \neq d) = \text{FALSE}$, $\text{val}(\text{NULL} = d) = \text{UNKNOWN}$, $\text{val}(\text{NULL} \leq d) = \text{UNKNOWN}$, etc.,

These boolean expressions are explicit RA selection conditions. Implicit selection conditions (exclusively equality between variables and/or constant) may also be expressed, for instance by joining two algebraic expressions with shared variables (natural join).

Leaf node types

Data node

Data nodes are prototypical leaf nodes of IQs. Intuitively, a data node is similar to a SPARQL triple pattern, but extended to n -uples.

Equivalently, a data node may represent the application to some $p \in P$ of a (possibly trivial) selection (σ in RA), followed by a variable renaming (ρ in RA).

Internally, a data node N is characterized (using logical notation) with an expression of the form $p(v_1 \sim, \dots, v_n \sim)$, where $p \in P$, and each $v_i \sim$ is either a constant or a variable, i.e. $v_i \sim \in D \cup V$. In addition, we have $\text{var}(N) = \{v_1 \sim, \dots, v_n \sim\} \cap V$.

Constants and repeated variable names express selection in the expected way.

For instance, if N is characterized by $p(x, x)$, then it represents the RA expression $\rho_{[x/p_1, x/p_2]}(\sigma_{[p_1=p_2]} p)$.

Therefore, if $I(p) = \{ \{p_1 \mapsto 1, p_2 \mapsto 2\}, \{p_1 \mapsto 3, p_2 \mapsto 3\} \}$, then $\|N\| = \{ \{p_1 \mapsto 3, p_2 \mapsto 3\} \}$ will hold.

Second order queries

SPARQL triple patterns allow for querying over the name of an RDF property, i.e. the name of a binary predicate. In an IQ, the data node corresponding to such a triple pattern is characterized by a ternary predicate with a meaningless predicate name (currently "triple"), and the property name variable as first argument. For instance, a data node encoding the SPARQL triple pattern $?s ?p ?o$ (where $?p$ stands for RDF property names) will be characterized by the expression $\text{triple}(p, s, o)$.

This mechanism may be extended in future versions to all predicates. In other words, P may be reduced to a set of one predicate per arity ("pair", "triple", "quadruple", etc), with the current $(n-1)$ ary predicate names as first argument.

Intensional VS extensional data nodes

Todo

Empty node

An empty node E can be viewed as a specific kind of data node (whether it should be considered as intensional or extensional is not important). The evaluation of the subquery rooted in E will always be (similar to) the RA n -ary empty relation, with $n \geq 0$.

In other words, for any database instance, $||E|| = \{\}$ holds (although $\text{var}(E)$ may be nonempty).

Alternatively, if $\text{var}(E) = \{v_1, \dots, v_n\}$, assuming some $p \in P$ with arity n , (the subtree whose only node is) E can be viewed as representing the RA expression $\rho_{[v_1/p_1, \dots, v_n/p_n]}(\sigma_{\text{FALSE}}(p))$.

The behavior of $||E||$ as an operand is similar to the behavior of an empty relation in RA. In particular (see [link:nonLeafNodes](#) [the dedicated sections] below for the definitions of the corresponding operators):

- (Left and right) identity element for union: $||E \text{ UNION } M|| = ||M||$, for any M s.t. $\text{var}(M) = \text{var}(E)$
- (Left and right) absorbing element for a natural join: $||E \text{ JOIN } M|| = ||E||$, for any M
- Left absorbing for left outer join, and right identity element if there is no join condition, or if the condition is met (right absorbing otherwise)
- Fixpoint for selection.

True node

True nodes represent another limit case. The evaluation of the subquery rooted in a true node T will always be (similar to) the RA nonempty but 0-ary relation. Or in other words, $\text{var}(T) = \{\}$, and for any database instance, $||T|| = \{f\}$, where f has an empty domain.

The behavior of $\llbracket T \rrbracket$ as an operand is in most cases similar to the behavior of its RA counterpart. In particular (see [link:nonLeafNodes](#)[the dedicated sections] below for the definitions of the corresponding operators):

- (Left and right) identity element for a natural join: $\llbracket T \text{ JOIN } M \rrbracket = \llbracket M \rrbracket$, for any M
- $\llbracket T \text{ UNION } T \rrbracket = \llbracket T \rrbracket$ (see the section about [link:bagSemantics](#)[bag semantics] for the treatment of cardinalities in this specific case).

As expected, T is also the right identity element for the left outer join operator:

- $\llbracket M \text{ LEFT JOIN } T \rrbracket = \llbracket M \rrbracket$, for any M .

In the case where T is the left operand, we have:

- $\llbracket T \text{ LEFT JOIN } M \rrbracket = \llbracket M \rrbracket$, if $\llbracket M \rrbracket \neq \{\}$
- $\llbracket T \text{ LEFT JOIN } M \rrbracket = \llbracket T \rrbracket$ otherwise.

Note that in RA, because the left outer join operator is traditionally defined in terms of join, projection, union and difference, these two queries would be syntactically invalid (implying a difference and a union of two relations with different arities). In IQs on the other hand, the left outer join is viewed as a primitive operator.

Non-leaf node types

Inner join node

An inner join node N represents the n -ary natural join of the evaluations of its children nodes, Explicit joining conditions can also be attached to it, in the form of a boolean expression $\varphi \sim N \sim$, as defined above.

It can be defined almost identically to its SPARQL counterpart. First, for any two partial functions f and g from V to D , let $f \sim g$ hold iff for all $x \in \text{dom}(f) \cap \text{dom}(g)$, we have $f(x) = g(x)$.

Then algebraic operation represented by an inner join node with explicit joining condition φ , denoted with $\text{JOIN}_{\sim\varphi\sim}$, is defined by:

Then $\|N_1 \text{ JOIN}_{\sim\varphi\sim} N_2\| = \{ f \cup g \mid (f,g) \in \|N_1\| \times \|N_2\| \text{ and } f \sim g \text{ holds and } \text{val}(\varphi[f \cup g]) = \text{TRUE} \}$, and $V(N_1 \text{ JOIN}_{\sim\varphi\sim} N_2) = v(N_1) \cup v(N_2)$.

If there is no explicit condition attached to an inner join node, then it represents the operation $N_1 \text{ JOIN}_{\sim 1=1\sim} N_2$, abbreviated as $N_1 \text{ JOIN } N_2$.

Left join node

A left join node represents the RA binary natural left outer join (or SPARQL OPTIONAL) of the evaluations of its children nodes. The left and right children nodes respectively stand for the left and right part of the join.

Explicit joining conditions may also be attached to it.

As for join nodes, this algebraic operation can be defined similarly to its SPARQL counterpart. An inner join node with explicit joining condition φ , represents the algebraic operation $\text{INNER JOIN}_{\sim\varphi\sim}$, defined by:

$\|N_1 \text{ INNER JOIN}_{\sim\varphi\sim} N_2\| = \|N_1 \text{ JOIN } N_2\| \cup \{ f \mid f \text{ is in } \|N_1\| \mid \text{for all } (g) \in \|N_2\|, f \sim g \text{ or } \text{val}(\varphi[f \cup g]) = \text{TRUE} \text{ does not hold} \}$.

As for join nodes, if there is no explicit condition attached to an inner join node, then it represents the operation $N_1 \text{ INNER JOIN}_{\sim 1=1\sim} N_2$, abbreviated as $N_1 \text{ INNER JOIN } N_2$.

Union node

Construction node

A construction represents a RA extended projection, followed by an attribute renaming.

"Extended" projection means here that some additional operations on the values of the projected attributes can be

expressed, prototypically string concatenation to create URIs out of data values (but also for instance arithmetic operations).

Attribute renaming is always the second operation, i.e. it is always performed on the output of the (extended) projection.

optional projection and renaming

Both operations (extended projection and renaming) are in a sense optional, in that the projection with all selected attributes (and no non-trivial additional operation) is the identity operator, and so is renaming with empty substitution. So in practice, a construction node may encode one of these operations.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Notation](#)

[Intermediate query optimization →](#)

Research

Research team

Publications

Theory

Notation

Formal characterization of IQs

Intermediate query optimization

Notation

Propagating down boolean expressions

Intermediate query optimization

WARNING

TODO: UPDATE AND CHECK (2 years old, imported from AsciiDocs)

This section provides a high level description of different transformations which may be applied to [Intermediate Queries](#) (IQs in what follows) for optimization purposes.

Notation

The different types of nodes of an IQ are presented [in the dedicated section](#). The following notation will also be use in the current document.

(Sub)trees

If s is a tree, then $\text{root}(s)$ will designate its root.

If n is a node in an IQ Q , then $\text{children}_{\sim Q}(n)$ (resp. $\text{children}(n)$ when Q is obvious from the context) will designate the immediate children of n in Q , and $\text{subtree}_{\sim Q}(n)$ (resp. $\text{subtree}(n)$) the maximal subtree of Q rooted in n . In addition, a "subtree of Q " will be understood as a maximal subtree of Q rooted in some node of Q . Finally, $\text{childSubtrees}_{\sim Q}(n)$ (resp. $\text{childSubtrees}(n)$) will stand for $\{\text{subtree}_{\sim Q}(c) \mid c \in \text{children}_{\sim Q}(n)\}$

Projected variables

If s is a subtree in an IQ, then $\text{var}(s)$ will designate the *variables*

projected out by s, defined inductively as follows:

- If root(s) explicitly projects out variables (construction node, union node, data node or empty node), then $\text{var}(s)$ is the set of such variables.
- Otherwise, $\text{var}(s) = \bigcup \sim s \in \text{childSubtrees}(n) \sim \text{var}(s)$

If e is an (explicit) join or filter condition, then $\text{var}(e)$ will designate the variables appearing in e .

Propagating down boolean expressions

This optimization is used for (some) NoSQL underlying DBMS. It consists in transforming an input IQ into an equivalent one such that, whenever possible, selections precede other algebraic operations. Or in other words, it consists in propagating (explicit) boolean expressions down the algebraic tree.

Three types of IQ nodes support boolean expressions, namely:

- filter nodes,
- inner join nodes, and
- left join nodes

For each node n of one of these types, if n has an explicit boolean expression e attached to it, then e is put into DNF, and each resulting conjunct may be propagated down independently. For instance, n may be a filter node, and e the expression $(x > y) \wedge (y \neq z)$. Then $(x > y)$ and $(y \neq z)$ may both be propagated down, independently from each other. If n is a node of one of the three types just mentioned, then $\text{bool}(n)$ will designate the set of such conjuncts.

Propagation decision and recipient selection

Propagation decisions are primarily taken based on a comparison between:

- the variables $\text{var}(e)$ present in the conjunct e being propagated down, and

- the variables $\text{var}(s)$ projected out by each candidate subtree s for the propagation of e .

Default case (inner join of filter provider node)

Let $e \in \text{bool}(n)$. The default rule to decide whether e can be propagated down from n is the following: for each $s \in \text{childSubtrees}(n)$, if $\text{var}(e) \subseteq \text{var}(s)$, then e can be propagated down to s .

An expression e being propagated down a subtree s also needs (a) new recipient node(s) to support it. The recipients selection procedure is the following:

- if $\text{root}(s)$ is neither a union node, construction node or left join node, then it is the (only) recipient of e in s .
- Otherwise, each $s' \in \text{childSubtrees}(\text{root}(s))$ recursively becomes a new candidate subtree for propagation. If there no $s' \in \text{childSubtrees}(\text{root}(s))$ to which e can be propagated down, then $\text{root}(s)$ is the (only) recipient of e in s . Note that in a valid IQ, if s is a construction or union node, then $\text{var}(s) \subseteq \text{var}(s')$ must hold for each $s' \in \text{childSubtrees}(\text{root}(s))$, therefore $\text{var}(e) \subseteq \text{var}(s')$ holds by transitivity, such that $\text{root}(s)$ cannot be the recipient for e in s . But it may be the case if s is a left join node.

Finally, if a recipient node r for e in s (or s') does not natively supports non-conditional boolean expressions, i.e. if r is neither a filter nor an inner join node, then a new filter node is created as the (immediate) parent of r , in order to support e . Alternatively (as a form of syntactic sugar), if the node n initially providing e is the immediate parent of r , then this new filter node is not needed, and e can remain attached to n , which in this case plays the role of "pseudo-recipient" of e in s .

Note that if n is the pseudo-recipient of e in s , then from the above selection procedure, $\text{root}(s)$ must be a data node, empty node, true node, or a left join node with no child accepting e , which in all cases guarantees that e cannot be further propagated

down s .

Specific case (left join provider node)

The propagation decision and recipient selection procedures are identical for a boolean conjunct e provided by a left join node n , but with an exception: e may only be propagated down the subtree rooted in the right child of n . Indeed, if e was propagated down the left subtree, an additional selection would be added to the algebraic expression represented by that left branch, violating the semantics of conditions attached to a left join (see [link:intermediateQuery_detailed.adoc#leftjoinNode](link:intermediateQuery_detailed.adoc#leftjoinNode)[the dedicated section] for an explanation).

Duplication decision

Let $e \in \text{bool}(n)$ for some node n in an IQ Q . And let us assume that there is at least one $s \in \text{childSubtrees}(n)$ to which e can be propagated, and that in each such s , a proper recipient node for e , different from n , has been found (or in other words that n is not the pseudo-recipient of e for any such s).

Then the decision still needs to be taken whether e should remain attached to n , (i.e. be duplicated down) or not (i.e. be strictly pushed down).

Default case

By default, an attempt is made to strictly push down e if it can be propagated down to at least one $s \in \text{childSubtrees}(n)$. So in the default case, and provided n is not a pseudo-recipient for e in any s , e will not remain attached to n .

Specific case (left-join-rooted candidate subtree)

If s is a candidate subtree for the propagation of e , and if a recipient node r in s has been found such that the path in Q from the provider node n of e to r comprises a left join node j and the right child j' of j , then e will remain attached to the provider node

n , i.e. e will be duplicated down, instead of strictly pushed down.

This prevents an alteration of the semantic of the IQ. Consider for instance the following example (where $r = j'$ for simplicity).

NOTE

Let:

- Q_1 be a well-formed IQ
- n_1 be a filter node, $n_1 \in Q_1$, and $\text{bool}(n_1) = \{e\}$, where e is the expression $x < 1$
- j_1 be a left join node with (explicit) joining condition $x \leq y$, and such that $\text{children}(n_1) = \{j_1\}$
- l_1 be the left child of j_1 , with $\text{var}(\text{subtree}(l_1)) = \{y\}$
- r_1 be a filter node and the right child of j_1 , with $\text{var}(\text{subtree}(r_1)) = \{x\}$

Then r_1 is a recipient for e .

Now consider the query Q_2 identical to Q_1 , where n_2 (resp. j_2 , r_2 , etc.) is the counterpart of n_1 (resp. j_1 , r_1 , etc.), but such that e has been pushed down to r_2 (and therefore is not attached to n_2 anymore). And let us assume a database instance defining an evaluation function $\|\cdot\|$, such that $\|l_1\| = \{ \{y \mapsto 2\} \}$ and $\|r_1\| = \{ \{x \mapsto 1\} \}$.

Then $\|j_1\| = \{ \{x \mapsto 1, y \mapsto 2\} \}$, and $\|n_1\| = \{ \{ \} \}$.

But $\|l_2\| = \{ \{y \mapsto 2\} \}$ and $\|r_2\| = \{ \{ \} \}$ must also hold, therefore $\|j_2\| = \{ \{y \mapsto 2\} \}$ and $\|n_2\| = \{ \{y \mapsto 2\} \}$, such that $\|n_1\| \neq \|n_2\|$.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Formal characterization of IQs](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Git workflow

In the development of Ontop project, we follow the `git-workflow`: <http://nvie.com/posts/a-successful-git-branching-model/>

A few points:

- Main development branch: `versionX`. Every commit in this branch is expected to be shipped with the next X.Y.Z release.
- Names of the branches:
 - feature branches: `feature/featurename`
 - bug fix branches: `bugfix/bugname`.
 - If there a corresponding GitHub page, use name `bugfix/bugname-issue-number` or `bugfix/issue-number`
 - Maintenance branches for releases: `releasing/vX.Y` (e.g. `releasing/v5.0`)
- Use `--no-ff` option for merging, especially for merging the feature or bugfix branches to `versionX`

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Git workflow

In the development of Ontop project, we follow the `git-workflow`: <http://nvie.com/posts/a-successful-git-branching-model/>

A few points:

- Main development branch: `versionX`. Every commit in this branch is expected to be shipped with the next X.Y.Z release.
- Names of the branches:
 - feature branches: `feature/featurename`
 - bug fix branches: `bugfix/bugname`.
 - If there a corresponding GitHub page, use name `bugfix/bugname-issue-number` or `bugfix/issue-number`
 - Maintenance branches for releases: `releasing/vX.Y` (e.g. `releasing/v5.0`)
- Use `--no-ff` option for merging, especially for merging the feature or bugfix branches to `versionX`

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
 - Update documents
 - Create a git branch for release
 - Update the version numbers
 - Build Maven packages
 - Build and distribute Ontop bundles
 - Update Docker Hub
 - Update the Ontop Website
 - Update the API example repository
 - Update branches for next development iteration
 - Time for celebration!
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Make a new release

Update documents

- `README.md` in `build/distribution`. This file will be uploaded to SourceForge and Github Releases
- `README.html` in `protege/plugin`. This file will be displayed in the Protégé Plugin Update window.
- `update.properties` file in `protege/plugin`. This is the configuration file for Protégé plugin Auto Update

Create a git branch for release

```
$ git checkout <main development branch>
$ git checkout -b releasing/<version-major.minor>
```

Update the version numbers

Update pom.xml files by Maven plugin to a SNAPSHOT version:

```
$ ./mvnw versions:set -DnewVersion=<version-major.minor>-SNAPSHOT
# $ ./mvnw versions:set -DnewVersion=5.0.0-SNAPSHOT
```

Build Maven packages

```
$ ./mvnw release:clean

# Preparing the release will sign artifacts (GPG key needed), create the new tag in git, and automatically push to github.
```



```
$ ./mvnw release:prepare -B

# stage the release
$ ./mvnw release:perform

# To skip tests:
# ./mvnw release:perform \
# -Darguments="-Dmaven.test.skip=true -DskipTests" \
# -Dgoals="deploy -Dmaven.test.skip=true -DskipTests"

# Or stage from a Git tag
# ./mvnw release:perform -DconnectionUrl=scm:git:git@github.com:ontop/ontop.git -Dtag=ontop-5.0.0
```

Stage the Maven artifacts at Maven Central

- URL: <https://central.sonatype.com/publishing/deployments>
- Click **Publish**

Build and distribute Ontop bundles

The bundles are located at

- CLI: `target/checkout/build/distribution/target/`
 - `ontop-cli-*.zip`
- Protege plugin and bundles:
`target/checkout/protege/distribution/target/`
 - `it.unibz.inf.ontop.protege-*.jar`
 - `ontop-protege-bundle-linux-*.tar.gz`
 - `ontop-protege-bundle-macos-*.zip`
 - `ontop-protege-bundle-win-*.zip`
 - `ontop-protege-bundle-platform-independent-*.zip`

We can also recompile them from the `target/checkout` directory generated during the Maven release

```
$ cd target/checkout
```

```
$ ./build-release.sh
```

Further info: [Build Instruction](#)

- Make a release in [Github Releases](#)
- Deploy the packages to [SourceForge](#) by manual upload or sftp.

```
$ sftp user,ontop4obda@frs.sourceforge.net
$ cd /home/pfs/project/o/on/ontop4obda
#
sftp://ghxiao,ontop4obda@frs.sourceforge.net/home/pfs/
```

Update Docker Hub

- Build and push the docker image of [ontop/ontop](#) to Docker Hub using the script `client/docker/build-image.sh`
 - by default, the script tag the images as `ontop/ontop:[version]` and `ontop/ontop:latest` - use option `-t` (one or more times) to tag differently (e.g., `-t ontop/ontop:xyz -t ontop/ontop:latest`)
 - run `client/docker/build-image.sh --help` for available options

```
# log in to Docker Hub
# docker login
$ cd ontop/client/docker/
# `-C`: multi-stage build. slow, but safe
# `-x`: compile for both amd64 and arm64, which entails pushing to Docker hub
$ ./build-image.sh -C -x
```

Update the Ontop Website

- Update [the guide homepage](#)
- Update [the release notes](#)

Update the API example repository

- Update the Ontop version in <https://github.com/ontop/ontop-api-examples>

Update branches for next development iteration

- Merge the tag for the release to master and the release branch to develop

```
$ git checkout develop
$ git merge release/v-number #branch

# this step is also for triggering the AutoUpdate
of Protege plugin
$ git checkout master
$ git merge v-number # tag
```

Time for celebration!

- Send release emails to several mail lists and social media

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Build Ontop Bundles](#)

[Build snapshots →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
 - Required implementations
 - Optional implementations
 - Declaring an implementation
 - Preparing and Running the Testcases
 - Further Implementations and Troubleshooting
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Support a new database system

This page describes how Ontop can be extended to support a novel relational database system (RDBMS).

The following RDBMSs are currently supported:

- DB2
- MySQL / MariaDB
- Oracle
- PostgreSQL
- SQL server
- H2
- Denodo
- Dremio
- Apache Spark SQL & Databricks
- Teiid
- Snowflake
- Trino
- Presto
- Athena
- Redshift
- DuckDB

It is relatively easy though to extend Ontop's source code in order to support an additional RDBMS, thanks to dependency injection.

New: Guided Implementation

Ontop also provides a [python tool](#) that automatically generates a large part of the required code end simplifies the process of developing support for new SQL dialects.

Using it minimizes the overhead for the implementation

of new dialect support and allows you to test your code easily right away.

This page provides a detailed explanation of the key concepts that need to be implemented, and was updated with Ontop 5.0.2. For newer versions of Ontop, it may no longer be up-to-date.

Required implementations

Two implementations must be provided, each for a different interface that already has a default (abstract) implementation.

These implementations dictate what the datatypes of the RDBMS are and how certain function symbols can be translated into the SQL dialect supported by the RDBMS.

The default implementation of each of these interfaces is often sufficient to handle many query operators and functions. As a result, only a few methods generally need to be overwritten, to account for specificities of the new SQL dialect.

The 2 required implementations are the DB function symbol factory and the DB datatype factory.

DB function symbol factory

The *DB function symbol factory* provides function symbols which can be directly serialized into the target SQL dialect (e.g. `LENGTH` or `CURRENT_TIMESTAMP`). This factory has methods for common operations with precise semantics (e.g. concatenating 3 string arguments in a null-rejecting manner).

The interface to implement is `DBFunctionSymbolFactory`. And the default implementation is `AbstractSQLDBFunctionSymbolFactory`.

For instance, the DB function symbol factory for PostgreSQL within Ontop is the class `PostgreSQLDBFunctionSymbolFactory`.

WARNING

This interface should not be confused with the `FunctionSymbolFactory`, which is in charge of constructing SPARQL function symbols and other function symbols that cannot be directly serialized into SQL.

A basic template for this implementation is provided here:

```

java
public class NewDialectDBFunctionSymbolFactory
    extends AbstractSQLDBFunctionSymbolFactory {

    @Inject
    protected
    DuckDBDBFunctionSymbolFactory(TypeFactory
    typeFactory) {

        super(createNewDialectRegularFunctionTable(typeFactory
        typeFactory));
    }

    protected static ImmutableTable<String,
    Integer, DBFunctionSymbol>
    createNewDialectRegularFunctionTable(TypeFactory
    typeFactory) {
        DBTypeFactory dbTypeFactory =
        typeFactory.getDBTypeFactory();
        DBTermType abstractRootDBType =
        dbTypeFactory.getAbstractRootDBType();

        Table<String, Integer, DBFunctionSymbol>
        table = HashBasedTable.create(

        createDefaultRegularFunctionTable(typeFactory));

        /*
        -----
        -
        -           Changed Function Symbols
        -
        -----
        */
    }

```

```
        return ImmutableTable.copyOf(table);
    }

    /**
     * -----
     * - Implementations -
     * -----
     */
}
```

In the template above, two code blocks are commented out. We will now look at each of them individually.

Changed Function Symbols

The *regular function table* is a dictionary, that maps from function names to the function symbol instance it corresponds to. The abstract base classes we inherit from already define a default function symbol table that includes all commonly used SQL functions. In some cases, however, some of these default function symbols need to be changed. For these cases, new entries can be created in this block, to override existing, bad entries.

A popular example is the `CURRENT_TIMESTAMP` function symbol. By default, it is created as a simple function with arity 0 of the form `CURRENT_TIMESTAMP()`. In some dialects, however, the current time is called without parentheses, simply in the form `CURRENT_TIMESTAMP`. In this case, we need to override the function symbol created by the default implementation, to not include parentheses:

```
DBFunctionSymbol nowFunctionSymbol = new
WithoutParenthesesSimpleTypedDBFunctionSymbolImpl(
    CURRENT_TIMESTAMP_STR,
    dbTypeFactory.getDBDateTimestampType(),
    abstractRootDBType);
table.put(CURRENT_TIMESTAMP_STR, 0,
nowFunctionSymbol);
```

The `CURRENT_TIMESTAMP` function is now a `WithoutParenthesesSimpleTypedDBFunctionSymbol`, so when generating SQL queries, ontop will not include the parentheses. By putting this new function symbol into the table at the key `CURRENT_TIMESTAMP_STR`, we override the default behaviour that was put into the table by the default method.

Implementations

A set of serialization methods have been left abstract by the base classes. These *always* have to be implemented for each new dialect.

All such methods share the same structure:

```
java
protected String
serialize[FUNCTION_NAME](ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    ...
}
```

Here, `terms` is the list of arguments that were passed to the function, `termConverter` is a function that can be used to transform any given term into its SQL equivalent, and `termFactory` can be used to generate new terms that may be required for some functions.

We will now go through each of them and provide example implementations.

Contains:

This function returns a boolean value, indicating whether a given string (argument 2) is included in a different string (argument 1). While some dialects have their own `CONTAINS` functions, for others it may have to be implemented through a string search, like in this example:

```
java
@Override
protected String serializeContains(ImmutableList<?
```



```
extends ImmutableTerm> terms,
Function<ImmutableTerm, String> termConverter,
TermFactory termFactory) {
    return String.format("(POSITION(%s IN %s) >
0)",

termConverter.apply(terms.get(1)),

termConverter.apply(terms.get(0)));
}
```

StrBefore:

This function returns the section of a given string (argument 1) before the first appearance of a given search string (argument 2). If the searched string is not contained in the base string, an empty string is returned instead. In many languages, this can be implemented using string search and substring methods.

```
@Override
protected String
serializeStrBefore(ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    String before =
termConverter.apply(terms.get(1));

    return
String.format("SUBSTRING(%s,1,POSITION(%s IN
%s)-1)", str, before, str);
}
```

StrAfter:

The opposite of `STRBEFORE`, this function returns the section of a given string (argument 1) *after* the first appearance of a search string (argument 2). This function can be implemented similarly:

```
@Override
protected String
serializeStrBefore(ImmutableList<? extends
```

```
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    String before =
termConverter.apply(terms.get(1));

    return String.format("SUBSTRING(%s,
IF(POSITION(%s IN %s) != 0, POSITION(%s IN %s) +
LENGTH(%s), 0))", str, after, str, after, str,
after);
}
```

Tz: This function extracts the timezone part of a given timestamp. An example implementation may look like this:

java

```
@Override
protected String serializeTz(ImmutableList<?
extends ImmutableTerm> terms,
Function<ImmutableTerm, String> termConverter,
TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    return String.format("
(LPAD(EXTRACT(TIMEZONE_HOUR FROM %s)::text,2,'0')
|| ':' || LPAD(EXTRACT(TIMEZONE_MINUTE FROM
%s)::text,2,'0'))", str, str);
}
```

DateTimeNorm: This function converts a given timestamp to the standardized `ISO 8601` format. An example implementation may look like this:

java

```
@Override
protected String
serializeDateTimeNorm(ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    return String.format("TO_IS08601(%s)",
termConverter.apply(terms.get(0)));
}
```

Hashing Functions:

Ontop supports a set of hasing functions:

- MD5
- SHA1
- SHA256
- SHA384
- SHA512

For each of them, the implementation of a serialization method is required. An example implementation of the MD5 function may look like this:

```
java
@Override
protected String serializeMD5(ImmutableList<?
    extends ImmutableTerm> terms,
    Function<ImmutableTerm, String> termConverter,
    TermFactory termFactory) {
    return String.format("MD5(%s)",
        termConverter.apply(terms.get(0)));
}
```

Other than the serialization methods, four additional methods must be implemented by the subclass:

```
java
@Override
protected String getUUIDNameInDialect() { return "
    [UUID_FUNCTION_NAME]"; }
```

returns the string that represents the name of the UUID in the SQL dialect.

The methods `createNullRejectingDBConcat(int arity)`, `createDBConcatOperator(int arity)`, and `createRegularDBConcat(int arity)` need to be implemented to define the concat function symbols. The `DBConcat` methods define function symbols that represent `CONCAT` function calls of a given arity. Sample implementations for them may look like this:

```
java
```

```
@Override
protected DBConcatFunctionSymbol
createRegularDBConcat(int arity) {
    return new
    NullToleratingDBConcatFunctionSymbol("CONCAT",
    arity, dbStringType, abstractRootDBType, false);
}

@Override
protected DBConcatFunctionSymbol
createNullRejectingDBConcat(int arity) {
    return new
    NullRejectingDBConcatFunctionSymbol("CONCAT_NO_NULL",
    arity, dbStringType, abstractRootDBType, false);
}
```

The `createDBConcatOperator(int arity)` method represents a concat operation executed through a concat operator (often `+` or `||`). It may be implemented like this:

```
@Override
protected DBConcatFunctionSymbol
createDBConcatOperator(int arity) {
    return new
    NullRejectingDBConcatFunctionSymbol("||", arity,
    dbStringType, abstractRootDBType,
    Serializers.getOperatorSerializer("||"));
}
```

java

DB Datatype factory

The *DB datatype factory* declares the hierarchy of DB datatypes used by the DBMS, and specifies their correspondence with datatypes used in the RDF graphs (such as xsd datatypes).

The interface to implement (for SQL dialects) is

```
SQLDBTypeFactory
```

And the default implementation is `DefaultSQLDBTypeFactory`.

For instance, the datatype factory for PostgreSQL within Ontop is the class `PostgreSQLDBTypeFactory`.

A basic template for this implementation is provided here:

java

```
public class NewDialectDBTypeFactory extends
DefaultSQLDBTypeFactory {

    /*
        -----
        -           Type Definitions           -
        -----
    */

    @AssistedInject
    protected NewDialectDBTypeFactory(@Assisted
TermType rootTermType, @Assisted TypeFactory
typeFactory) {

super(createNewDialectTypeMap(rootTermType,
typeFactory), createNewDialectCodeMap());
    }

    protected static Map<String, DBTermType>
createNewDialectTypeMap(TermType rootTermType,
TypeFactory typeFactory) {
        /*
            -----
            -           Type Map           -
            -----
        */
    }

    protected static ImmutableMap<DefaultTypeCode,
String> createNewDialectCodeMap() {
        /*
            -----
            -           Code Map           -
            -----
        */
    }

    /*
```

```
-----  
-                               Support Flags                               -  
-----  
  
    */  
  
}
```

In the template above, four code blocks are commented out. We will now look at each of them individually.

Type Definitions

In this section, we define a set of `String` constants that represent the names of datatypes in our SQL dialect. These constants are then used in other code blocks. An excerpt of the *type definitions* part in an implementation may look like this:

```
protected static final String BIT_STR = "BIT";  
protected static final String INT2_STR = "INT2";  
protected static final String INT4_STR = "INT4";  
protected static final String INT8_STR = "INT8";  
protected static final String FLOAT4_STR =  
    "FLOAT4";  
protected static final String FLOAT8_STR =  
    "FLOAT8";
```

TYPE MAP

In this method, we define the contents of the type map for our dialect. We create `DBTermType` instances for each supported type and store them in a `Map` object, that matches their identifiers to the `DBTermType`. The base class we extend from already generates an extensive type map for many popular types, such as `VARCHAR`, `INTEGER`, `REAL`, `DOUBLE`, and more in the method `createDefaultSQLTypeMap(...)`. New types can be added to the map in the subclass, and, alternatively, we can override existing entries in the map here. An example implementation of this method may look like this:

```
protected static Map<String, DBTermType>
```

```
createNewDialectTypeMap(TermType rootTermType,
TypeFactory typeFactory) {
    TermTypeAncestry rootAncestry =
rootTermType.getAncestry();

    Map<String, DBTermType> map =
createDefaultSQLTypeMap(rootTermType,
typeFactory);

    return map;
}
```

CODE MAP

The code map is a dictionary, that maps different type codes to their "default type" for the SQL dialect. This includes the following type codes:

- `STRING`
- `HEXBINARY`
- `LARGE_INTEGER`
- `DECIMAL`
- `DOUBLE`
- `BOOLEAN`
- `DATE`
- `TIME`
- `DATETIMESTAMP`
- `GEOMETRY`
- `GEOGRAPHY`
- `ARRAY`
- `JSON`

The default implementation already maps many of these to the name of the corresponding data type. To support additional ones, or to override existing mappings, this method can be implemented. An example implementation may look like this:

```
protected static ImmutableMap<DefaultTypeCode,
String> createNewDialectCodeMap() {
    Map<DefaultTypeCode, String> map =
createDefaultSQLCodeMap();
}
```

java

```
map.put(DefaultTypeCode.JSON, JSONB_STR);

return ImmutableMap.copyOf(map);
}
```

Support Flags

An instance of `DBTypeFactory` supports a set of methods that, when called, return a flag indicating if a certain SQL feature is supported by the dialect. These methods are

- `supportsDBGeometryType()`
- `supportsDBGeographyType()`
- `supportsDBDistanceSphere()`
- `supportsJson()`
- `supportsArrayType()`

Any of these functions can be overridden by the `DBTypeFactory` sub-class to return a boolean constant indicating if the type is supported.

Other Implementations

The `DBTypeFactory` base-class supports further methods that can be included in sub-classes for fine-tuning. One popular set of such methods are the `LexicalValue` methods. These return a `String` constant that represents the lexical value the corresponding literal may assume. For instance, by default, the method `getDBFalseLexicalValue()` returns the string `"FALSE"`, as this is a common representation of the boolean `false` value in many SQL dialects. To implement the `DBTypeFactory` for a dialect that instead uses the digit `0` for `false` literals, we could override the method like so:

```
@Override
public String getDBFalseLexicalValue() {
    return "0";
}
```

java

Optional implementations

Additional implementations can be optionally provided in replacement of the default implementation.

Serializer

The *serializer* dictates the general shape (SELECT-FROM-WHERE) of the SQL query.

The interface to implement is `SelectFromWhereSerializer`.
And the default implementation is `DefaultSelectFromWhereSerializer`.

For instance, the serializer for PostgreSQL within Ontop is the class `PostgresSelectFromWhereSerializer`.

Normalizer

The *normalizer* addresses limitations of certain DBMSs (such as a non-canonical evaluation of the ORDER BY clause).

The interface to implement is `DialectExtraNormalizer`.
And several implementations are already available, some of which are used by several DBMS.

For instance, the normalizer associated to PostgreSQL within Ontop is

`OnlyInPresenceOfDistinctProjectOrderByTermsNormalizer`.

Metadata provider

The *metadata provider* specifies how schema and integrity constraints (for instance primary keys) are retrieved from the DBMS.

The interface to implement is `MetadataProvider`.
And the default implementation is `DefaultDBMetadataProvider`.

For instance, the metadata provider for PostgreSQL within Ontop is `PostgreSQLDBMetadataProvider`.

Declaring an implementation

All the implementations mentioned above can be declared in the property file `sql-default.properties` (which can be found in the directory `it/unibz/inf/ontop/injection/` of the `ontop-rdb` module).

A key-value pair must be added for each of these implementations, where the key indicates the type of the implementation (serializer, function symbol factory, etc.), and the value is the implementation.

The naming scheme for the keys is the following.

Let `<driverName>` be the name of the JDBC driver for the RDBMS (for instance, the JDBC driver for PostgreSQL is `org.postgresql.Driver`).

Then the keys are:

- `<driverName>-serializer` for a serializer
- `<driverName>-symbolFactory` for a DB function symbol factory
- `<driverName>-typeFactory` for a DB datatype factory
- `<driverName>-normalizer` for a normalizer
- `<driverName>-metadataProvider` for a metadata provider

For instance, the key-value pairs declared for PostgreSQL are:

```
org.postgresql.Driver-serializer =
it.unibz.inf.ontop.generation.serializer.impl.Postgres

org.postgresql.Driver-symbolFactory =
it.unibz.inf.ontop.model.term.functionsymbol.db.impl.P

org.postgresql.Driver-typeFactory =
it.unibz.inf.ontop.model.type.impl.PostgreSQLDBTypeFac

org.postgresql.Driver-normalizer =
it.unibz.inf.ontop.generation.normalization.impl.OnlyI
```

```
org.postgresql.Driver-metadataProvider =
it.unibz.inf.ontop.dbschema.impl.PostgreSQLDBMetadataP
```

A basic set of key-value pairs may be:

```
name.of.jdbc.Driver-serializer =
it.unibz.inf.ontop.generation.serializer.impl.DefaultS

name.of.jdbc.Driver-symbolFactory =
it.unibz.inf.ontop.model.term.functionsymbol.db.impl.N

name.of.jdbc.Driver-typeFactory =
it.unibz.inf.ontop.model.type.impl.NewDialectDBTypeFac

name.of.jdbc.Driver-normalizer =
it.unibz.inf.ontop.generation.normalization.impl.OnlyI

name.of.jdbc.Driver-metadataProvider =
it.unibz.inf.ontop.dbschema.impl.DefaultSchemaCatalogD
```

Preparing and Running the Testcases

Ontop currently uses a set of tests called `lightweight-tests` to test all main ontop language support features on minimal databases. A simple way of testing the correctness of newly implemented dialect support is to run the same tests a database that uses the corresponding dialect.

This consists of the following steps:

1. Set up a database with the required testing conditions
2. Prepare files that assist the connection to the database
3. Implement sub-classes of the abstract lightweight test classes and run them

In the following sections, we will discuss each of these steps.

Setting up the Database

The exact procedure of the database preparation depends on the database system. In ontop, we generally use docker images that reproduce the exact same database state for each test run so we can test the different dialects automatically.

Each directory inside `test/lightweight-tests/lightweight-db-test-images/` corresponds to one docker image, and many of them contain table definitions in `.sql` file. For instance, the `mysql` directory contains the exact definitions of the three databases that are used for testing, `books`, `dbconstraints`, and `university`.

To prepare the required testing conditions, you can take these `.sql` files and run their queries on your database. Should any of the features not be supported, you will be required to either use alternative equivalent features to reach the same final database state, or disable the corresponding test cases.

Preparing Connection Files

The directory `test/lightweight-tests/src/test/resources/` contains files that are used for setting up an ontop connection with different databases, including `.obda` files for the VKG creation and `.property` files for JDBC connections. While the `.obda` files can usually be reused for different systems, the `.property` files must be included for each database system. The directories `books` and `prof` inside the test resources contain one sub directory for each dialect, where each sub directory in turn includes one `.properties` file for the JDBC connection. The `dbconstraints` and `university` directories, on the other hand, directly contain `.property` files for each dialect.

To run all lightweight tests for a new dialect, the corresponding files and directories have to be created for its JDBC connection. In particular, the connection files should be created in such a way, that the corresponding database/schema of the test is selected as the default database/schema, as the `.obda` files access the

tables through relative paths.

Implementing the Lightweight Test Classes

The directory `test/lightweight-tests/src/test/java/` contains the test cases that have to run successfully for each new dialect. They are defined in the four abstract classes

`AbstractBindTestWithFunctions`, `AbstractConstraintTest`, `AbstractDistinctInAggregateTest`, and `AbstractLeftJoinProfTest`.

For each new dialect, a new package has to be created at this location, containing the implementations of sub-classes for each of these abstract base classes.

`BindTestWithFunctions`

This test class tests all main SQL functions that are supported by ontop. It uses the `books` database, as well as directly generated data through the SPARQL `BIND` function. An example implementation of this class may look like this:

```
java
@NewDialectLightweightTest
public class BindWithFunctionsNewDialectTest
extends AbstractBindTestWithFunctions {

    private static final String PROPERTIES_FILE =
        "/books/new-dialect/books-new-dialect.properties";

    @BeforeAll
    public static void before() throws
        IOException, SQLException {
        initOBDA(OBDA_FILE, OWL_FILE,
            PROPERTIES_FILE);
    }

    @AfterAll
    public static void after() throws SQLException
    {
        release();
    }
}
```

```

    /*
       -----
       -               Result Definitions               -
       -----
    */

}

```

NOTE

Using the `@NewDialectLightweightTest` annotation requires the definition of the annotation in the same package, similarly to how it was defined for other dialects.

The code block "Result Definitions" is commented out in the above example. This code block may contain redefinitions of expected results that are tested in the base class. To do this, override the `get[TEST-NAME]ExpectedValues()` method here. An example implementation may look like this:

```

@Override
protected ImmutableList<String>
getConstantIntegerDivideExpectedResults() {
    return
    ImmutableList.of("\"0.500\"^^xsd:decimal");
}

```

Should any of the functionalities not be supported by the SQL dialect, then the corresponding test case may simply be disabled with an explanation as to why it is not supported. An example for that may look like this:

```

@Disabled("New Dialect does not support SHA384")
@Test
@Override
public void testHashSHA384() {
    super.testHashSHA384();
}

```

Some test cases are disabled by default.

Constraint

This set of test cases tests for the correct interpretation of integrity constraints in the data. It uses the `dbconstraints` database. An example implementation may look like this:

```
java
@NewDialectLightweightTest
public class ConstraintNewDialectTest extends
    AbstractConstraintTest {

    private static final String PROPERTIES_FILE =
        "/dbconstraints/dbconstraints-new-
        dialect.properties";

    public ConstraintNewDialectTest(String method)
    {
        super(method, PROPERTIES_FILE);
    }

}
```

Once again, specific tests may be disabled if they are not supported. For database systems that do not support integrity constraints, this class can be ignored.

DistinctInAggregate

This set of test cases runs queries containing aggregate functions that include the `DISTINCT` keyword. It can be implemented analogously to the `BindWithFunctions` test class.

LeftJoinProf

This set of test cases runs on the `university` database. It accesses items that have to be taken from the database through `JOIN`s. Furthermore, it tests the optimization procedures employed by ontop to minimize the required number of joins. Once again, it can be implemented analogously to the `BindWithFunctions` test class.

If the database management system does not support integrity constraints, many of these test cases will fail because the optimizations will not be performed.

Further Implementations and Troubleshooting

Once the basic features and lightweight test cases have been implemented for a dialect, the tests can be run to determine which features still do not work correctly.

If a test case fails, its corresponding feature is likely not yet supported by the current implementation. In these cases, changes need to be applied to the implemented `DBTypeFactory` or `DBFunctionSymbolFactory` to solve the issue.

In some situations, however, issues may be caused by problems not related to database types or functions. In these cases, one of the following default implementations may also be extended for the new dialect.

NOTE

New implementations of these classes have to be linked to the driver through the `sql-default.properties` file.

```
DefaultSchemaCatalogDBMetadataProvider,  
DefaultSchemaDBMetadataProvider, or  
DefaultDBMetadataProvider
```

If the `DBMetadataProvider` assigned to the dialect is not able to perform its expected actions through its default implementation, it may have to be extended with a new sub-class that is particularly tailored for this dialect.

In that case, its methods may be overridden to change the default behaviour. Common targets include:

`insertIntegrityConstraints(...)` if the dialect does not support integrity constraints.

- `getRelationId(...)` to return the ID of a given relation.

Additionally, the constructor may be implemented to determine the default schema or default catalog names if they cannot be accessed directly through the JDBC, or a different

`QuotedIDFactory` instance can be passed to the super-class to force ontop to generate quoted identifiers in a specific way.

SelectFromWhereSerializer

This class is responsible for the serialization of the SQL query frame. Different SQL dialects have different rules as to how the general shape of an SQL query have to be defined. For instance, some dialects allow users to combine `LIMIT` and `OFFSET` clauses as `LIMIT <offset>,<limit>`, while others require the form `OFFSET <offset> LIMIT <limit>`. The following shows a default implementation of a `SelectFromWhereSerializer` to achieve the latter behaviour:

```
java
@Singleton
public class NewDialectSelectFromWhereSerializer
    extends DefaultSelectFromWhereSerializer
    implements SelectFromWhereSerializer {

    @Inject
    private
    NewDialectSelectFromWhereSerializer(TermFactory
    termFactory) {
        super(new
        DefaultSQLTermSerializer(termFactory));
    }

    @Override
    public QuerySerialization
    serialize(SelectFromWhereWithModifiers
    selectFromWhere, DBParameters dbParameters) {
        return selectFromWhere.acceptVisitor(
            new
            DefaultRelationVisitingSerializer(dbParameters.getQuot
```

```
        @Override
        protected String
serializeLimitOffset(long limit, long offset,
boolean noSortCondition) {
            return
String.format("OFFSET %d LIMIT %d", offset,
limit);
        }
    });
}
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Debug Jetty](#)

[Statistics →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
 - Build the Protégé plugin
 - Debug in IntelliJ using the Remote debugger
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Debug Protégé

This page describes how to debug the Protégé Plugin of Ontop from IntelliJ.

If you just want to test the plugin (without debugging), you only need to follow the step of building the Protégé plugin, and then run Protégé as usual (use `run.bat` on Windows)

```
$ Protege-[protege-version]/run.sh
```

Build the Protégé plugin

Requirements

Checkout and build Ontop source code as described [here](#). We assume all commands listed below are issued from the root directory of Ontop source tree.

First time build

Unzip the Ontop Protégé bundle (obtained by previously building Ontop source code), which contains Protégé preconfigured with the Ontop Protégé plugin

```
$ unzip protege/distribution/target/ontop-protege-bundle-platform-independent-[ontop-version].zip
```

Notes:

- for older versions of Ontop (before 5.0.0) the bundle to unzip is located either in `build/distribution/ontop-protege` (up to 4.0.3) or in `build/distribution/target` (up to 5.0.0 excluded);

- you may also reuse a (platform independent) Protégé distribution obtained directly from Protégé web site and already installed locally. In this case make sure that the Protégé version matches the one used during Ontop build, that Protégé directory contains a `plugins` sub-directory (create it if missing), and run the commands below to deploy the Ontop Protégé plugin (adapting paths accordingly).

Subsequent builds

Each time you modify the code, you need to recompile and build the Ontop distribution.

The fastest command to re-compile the Ontop Protégé plugin and the code it depends on is

```
$ mvn clean package -Dmaven.test.skip -Passet-plugin -pl protege/distribution -am
```

Where:

- `clean package` forces a full re-build and may be generally omitted (which defaults to running `package`) unless you encounter build errors, in which case `clean` helps ruling out they stem from incremental builds;
- `-Dmaven.test.skip` disables compiling and executing unit tests;
- `-Passet-plugin` activates the generation of the Ontop Protégé plugin (OSGI bundle with all its dependencies), without including other unneeded artifacts (e.g., full ZIP/tar.gz Protégé bundles, source and Javadoc JARs, etc);
- `-pl protege/distribution -am` (optional) tells Maven to only build the Ontop Protégé module located in `protege/distribution` and all the other Ontop modules it depends on, saving another couple of seconds.

Notes:

- in Ontop versions 4.1 to 5.0.0 excluded, the command above should be replaced by `mvn clean package -`

```
mvn test -Dmaven.test.skip -Pprotege -pl client/protege -am ;
```

- in Ontop version 4.0.3 or earlier, the command above should be replaced by `mvn clean install -pl client/protege -am -DskipTests` to compile code and install it in local maven repository, followed by `mvn bundle:bundle -pl client/protege` to package the plugin.

Once the Ontop Protégé plugin is built, it must be copied to the Protégé plugin directory via

```
$ cp
protege/distribution/target/it.unibz.inf.ontop.protege
[ontop-version].jar Protege-[protege-
version]/plugins/
```

The command above should replace any previous Ontop Protégé plugin in the `plugins` directory. In any case, make sure that there is only one Protégé plugin in `Protege-[protege-version]/plugins/`

Debug in IntelliJ using the Remote debugger

IntelliJ configuration

Create a remote configuration:

- select 'Edit configurations', click '+', then select 'Remote'
- Host: `localhost`
- Port: choose a port number (`5005` in what follows)

Protégé run options

Edit the script `Protege-[protege-version]/run.sh` (on Windows: `run.bat`), adding the following JVM option (note that parameter `suspend` is set to `y`)

```
-  
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,a
```

Run / Debug

Execute `Protege-[protege-version]/run.sh` (on Windows: `run.bat`). Then, from IntelliJ, click on the Debug button (or Shift+F9).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Build snapshots](#)

[Debug Jetty →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
 - Start the Jetty in Debug mode
 - Linking with IntelliJ
 - References
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Debug Jetty

This page describe debug Jetty with RDF4J workbench plugin in IntelliJ Idea

Start the Jetty in Debug mode

```
$ java -Xdebug -
agentlib:jdwp=transport=dt_socket,address=9999,server=
-jar start.jar
```

Linking with IntelliJ

- Next we need to link the IntelliJ project with the deployed webapp.
1. Within IntelliJ, open the project containing the webapp deployed into jetty that you want to debug. SelectRun -> Edit Configurations. Add a new configuration by clicking the "+" icon. Choose Remote. Make sure the port you choose is the same as the one you added in Enable remote debugging.
 2. Next in your webapp you can set a breakpoint within a servlet which when it is tripped will halt the remote jvm's processing thread to await for debugging commands from your IntelliJ instance. To set a breakpoint, simply open the servlet or any other class you want to debug and click left to the line you want to set the breakpoint at (where the red dot is on the next screenshot). The red dot and red background on the line mark the breakpoint.
 3. Accessing that servlet within your browser, pointed at your remote debug configured jetty-distribution, should transition

your IntelliJ instance to the standard debugger view.

References

- 1. <http://www.eclipse.org/jetty/documentation/current/enable-remote-debugging.html>
- 2. <http://www.eclipse.org/jetty/documentation/current/debugging-with-intellij.html>

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← [Debug Protégé](#)

[Support a new database system](#) →

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- IQ
- Query nodes
- Previous query format (version 1)
- Guice Conventions

Outdated corner ▶

Intermediate Query (IQ)

Input queries for Ontop are expressed in (a fragment of) SPARQL 1.1, whereas the output of the query reformulation process is expressed in the query language used by the underlying DBMS (prototypically some variant of SQL but also other query languages like the MongoDB Aggregate Framework).

Because most query rewriting and optimization steps apply regardless of the underlying DBMS, a DBMS-independent query representation format is used internally. This format is called *intermediate query* (IQ) and allows for the representation of queries and mapping assertions.

IQ

In terms of implementation, since 4.3.0, intermediate queries are always immutable and based on the class `IQ`. Prior to that, there was also a mutable variant, centered on the former class `IntermediateQuery`, which was actually the first implementation.

Query nodes

An IQ is a standard (rooted and ordered) tree representation of a algebraic query expression. In terms of implementation, it is based on a projection atom and an `IQTree`.

The underlying algebra is a compromise between (a fragment of) the SPARQL algebra on the one hand, and the select/project/join/rename/union relational algebra (RA in what follows) with named attributes on the other hand.

The following is an informal description of the different types of nodes which may appear in an IQ. A formal characterization (in

terms of RA algebra) is given in [the dedicated section](#).

Optimizations applied to IQs are described in [the dedicated section](#).

Leaf node types

Intensional data node

A intensional node is a prototypical leaf node that is expected to be replaced by an `IQTree` when unfolding the mapping.

It is characterized by a predicate ϕ , where each ϕ_i is either a ground term (constant or functional term without variable) or a variable name.

Intensional data nodes are typically used for representing triple and quad patterns from SPARQL.

Extensional data node

An extensional data node is similar to an intensional data node, expect that:

- It is not required to be unfolded (e.g. if it represents a database table)
- It has a sparse representation: only a subset of columns are mapped to variables or ground terms.

Empty node

An empty node can be viewed as a specific type of data node, whose evaluation will always be an empty set of tuples.

See [the dedicated section](#) for a precise characterization.

True node

True nodes represent another limit case, namely an expression whose evaluation will always be a singleton set containing the 0-ary tuple.

See [the dedicated section](#) for a precise characterization.

Native node

A native node is defined from a native query string (e.g. a SQL string) and explicits the datatypes of the variables returned by the native query. It is generated at the end of query reformulation. It is not used in the mapping.

Values node

Since 4.2.0. A Values node, similarly to the `VALUES` keyword in SQL and SPARQL, represents an "embedded table" having its rows embedded in the query.

Non-leaf node types

Filter node

A filter node represents an RA selection over the evaluation of its (unique) child node.

Inner join node

An inner join node represents the RA n-ary natural join of the evaluations of its children nodes.

Explicit joining conditions may also be attached to it.

See [the dedicated section](#) for a precise characterization.

Left join node

A left join node represents the RA binary natural left outer join of the evaluations of its children nodes. The left and right children nodes respectively stand for the left and right part of the join.

Explicit joining conditions may also be attached to it.

See [the dedicated section](#) for a precise characterization.

Union node

A union node represents the RA n-ary union of the evaluations of its children nodes, possibly preceded by a (unique) projection applied to each of them.

See [the dedicated section](#) for a precise characterization.

Construction node

A construction node represents a sequence of (at most) three operations applied to the evaluation of its unique child node:

- a projection, followed by
- the application of a function to the values of each retrieved tuple, followed by
- a variable renaming.

For instance, a construction node may first project out variables and , then prepend the string "http://domain/" to each value retrieved for , and finally rename variable with .

Each of these three operations is optional.

See [the dedicated section](#) for a precise characterization.

Aggregation node

An aggregation node is defined by the grouping variables over which the aggregation occurs (if any) and a substitution defined the new variables obtained by from aggregation functions.

It projects the grouping variables and the variables defined by the substitution.

Slice node

A slice node contains a limit and/or an offset value, that is uses for eliminate tuples from the result set.

Distinct node

A distinct node keeps only one occurrence of each tuple.

Order by node

An order by node orders the tuples based on a list of comparators. It orders first the results according to the first comparator (ascending or descending) and, in case of same ordering, it orders them using the following comparator, and so on.

This operator follows a **NULLS FIRST** semantics (if ascending, **NULLS LAST** otherwise), similarly to SPARQL.

Previous query format (version 1)

Datalog was the main internal query representation format in Version 1 of Ontop, due in particular to its close similarity with conjunctive queries, which were the only supported input queries. The switch to SPARQL input queries increased expressiveness, making their rule-based representation more complex (due in particular to the the SPARQL OPTIONAL operator, which, roughly speaking, corresponds to the SQL left outer join). The intermediate query format on the other hand offers a relatively straightforward internal representation, not only of SPARQL queries, but also of the queries sent to underlying DBMS (at least the SQL-like ones), as well as complex mapping assertions (which, in most scenarios, contains SQL queries). This is also why the rule-based internal representation has been progressively abandoned.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Statistics](#)

[Guice Conventions →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Guice Conventions

Each module contains a folder `src/main/java/` and a module `it.unibz.inf.ontop.injection.filename.properties` (e.g., `mapping-default.properties`).

The file specifies to which implementation each interface needs to be instantiated. For instance, `mapping-default.properties` contains the following lines

```
#####
# Default implementations
#####

it.unibz.inf.ontop.spec.OBDASpecificationExtractor
=
it.unibz.inf.ontop.spec.impl.DefaultOBDASpecificationE

it.unibz.inf.ontop.spec.mapping.transformer.MappingTra
=
it.unibz.inf.ontop.spec.mapping.transformer.impl.Defau

...

```

`DefaultOBDASpecificationExtractor` is an implementation for the interface `OBDASpecificationExtractor`. The constructor, private, is injected, that is it has the `@Inject` tag at its beginning.

```
@Inject
private
DefaultOBDASpecificationExtractor(MappingExtractor
mappingExtractor, MappingTransformer
mappingTransformer,

OntopMappingSettings settings) {
    this.mappingExtractor = mappingExtractor;
}

```

```

        this.mappingTransformer =
mappingTransformer;
    }
}

```

The `@Inject` means that the arguments to the constructor will be provided by the injection mechanism. For instance, the constructor for the second parameter is:

```

@Inject
    private
DefaultMappingTransformer(MappingVariableNameNormalize
mappingNormalizer,

MappingSaturator mappingSaturator,

ABoxFactIntoMappingConverter inserter,

MappingMerger mappingMerger,

OntopMappingSettings settings,

MappingSameAsInverseRewriter
sameAsInverseRewriter,

SpecificationFactory specificationFactory,
                                                                    RDF
rdfFactory,

MappingDistinctTransformer
mappingDistinctTransformer) {
    this.mappingNormalizer =
mappingNormalizer;
    this.mappingSaturator = mappingSaturator;
    this.factConverter = inserter;
    this.mappingMerger = mappingMerger;
    this.settings = settings;
    this.sameAsInverseRewriter =
sameAsInverseRewriter;
    this.specificationFactory =
specificationFactory;
    this.rdfFactory = rdfFactory;
    this.mappingDistinctTransformer =
mappingDistinctTransformer;
}

```

And a constructor, let's say, for `RDFFactory` is

```
public class SimpleRDF implements RDF {  
  
    public SimpleRDF() {  
    }  
    ...  
}
```

As you see, in the end we reach a constructor without parameters.

In `IntelliJ`, the name of the properties file binding a specific interface can also be retrieved through the `find usages` functionality.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Intermediate Query \(IQ\)](#)

[Known issues →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Build Ontop Bundles

Requirements

- a Linux, Mac OS X or Windows machine;
- git, for checking out Ontop code (refer to installation instructions for your OS/distribution);
- Java 11 (JDK, not JRE, refer to installation instructions for your OS/distribution);
- JAVA_HOME environment variable correctly set up and pointing to a Java 11 JDK. Run `echo $JAVA_HOME` (on Windows: `echo %JAVA_HOME%`) and check the path printed (if any). If the variable is unset, you can set it temporarily in the terminal session using `export JAVA_HOME=[path]` (on Windows: `setx JAVA_HOME -m "[path]"`), where [path] points to the JDK installation (the directory containing bin, lib, etc). This setting may be made persistent by editing your `~/.bash_profile` (on Windows, edit System and Security > System > Advanced System Settings > Environment Variables). In particular, for Mac OS X

```
$ vim ~/.bash_profile
Add this line: export
JAVA_HOME=$(/usr/libexec/java_home)
$ source ~/.bash_profile
```

- Maven 3.6.0 or later, with `mvn` command available on `PATH`. Run

```
$ mvn -v
Apache Maven 3.9.0
(9b58d2bad23a66be161c4664ef21ce219c2c8584)
Maven home: /opt/maven/current
```

```
Java version: 11.0.17, vendor: Debian, runtime:
/usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "6.1.0-3-amd64",
arch: "amd64", family: "unix"
```

Check the versions of Maven (3.9.0 > 3.6.0 in the example) and Java (11.0.17, i.e. Java 11).

Checkout Ontop source code

In a terminal, move to the parent directory where you want to checkout Ontop, and run

```
$ git clone https://github.com/ontop/ontop.git
$ cd ontop
```

Build all bundles

Starting from Ontop 4.1, from the root directory of Ontop source code tree, run

```
$ mvn -Dmaven.test.skip -Prelease
```

The command compiles Ontop modules and generates the Ontop bundles distributed on GitHub, whose files are written in two directories.

Under `build/distribution/target/`:

- `ontop-cli-[ontop-version].zip` - libraries/scripts to use ontop from the command line and setup a SPARQL endpoint;

Under `protege/distribution/target/`:

- `it.unibz.inf.ontop.protege-[ontop-version].jar` - plugin for Protégé 5.x installation files (drop it in Protégé's plugins folder);
- `ontop-protege-bundle-platform-independent-[ontop-
version].zip`

- `[version].zip` - Protégé 5.x bundled with Ontop plugin and ready to run (just unzip and use the `run.sh` or `run.bat` scripts);
- `ontop-protege-bundle-linux-[ontop-version].zip` - as above, for Linux;
- `ontop-protege-bundle-os-x-[ontop-version].zip` - as above, for Mac OS X;
- `ontop-protege-bundle-win-[ontop-version].zip` - as above, for Windows.

Notes:

- profile activation `-Prelease` enables the generation of all the Ontop bundles as well as source code and Javadoc JARs needed for publishing on Maven Central (if this profile is omitted, only JAR files for Ontop binaries are created);
- option `-Dmaven.test.skip` disables the compiling and execution of unit and integration tests (generally performed at release time) to speed up the process - omit it to enable tests, e.g., for testing changes to source code (note: may also use `-DskipTests` that disables executing tests, but still compiles them);
- the build may be performed also by running script `build-release.sh` (`build-release.cmd` on Windows), which works also in older Ontop versions (4.0.3 or earlier). In these older versions, the script generates bundles in sub-directories `ontop-cli`, `ontop-protege`, and `ontop-webapps` of directory `build/distribution`, and also generates bundles for Jetty 9 + RDF4J workbench + Ontop and for Tomcat 8 + RDF4J workbench + Ontop. These bundles have been deprecated in Ontop 4 and completely removed since Ontop 5 .

Build specific bundles

Starting from version `4.1` and further revised in version `5.0.2`, we provide Maven profiles to build specific Ontop bundles as quickly as possible (with respect to using the above 'build all' command), which may come in handy when modifying

Ontop source code. Specifically:

- to build the Ontop command line tool only, i.e., `ontop-cli-[ontop-version].zip` (note: up to version `5.0.1`, `asset-cli` was `cli` and tests were disabled by default)

```
$ mvn -Dmaven.test.skip -Passet-cli
```

- to build the Ontop Protégé plugin, i.e.,

```
it.unibz.inf.ontop.protege-[ontop-version].jar
```

(profile available since version `5.0.2`)

```
$ mvn -Dmaven.test.skip -Passet-plugin
```

- to build the Ontop Protégé plugin

```
it.unibz.inf.ontop.protege-[ontop-version].jar
```

as well as the `ontop-protege-bundle-[platform]-[ontop-version].zip` Protégé bundles including it (note: up to version `5.0.1`, `asset-protege` was `protege` and tests were disabled by default)

```
$ mvn -Dmaven.test.skip -Passet-protege
```

The Maven profiles `asset-cli`, `asset-plugin`, `asset-protege` employed in the commands above exclude unneeded artifacts from the build, which is thus faster. These profiles can be arbitrarily combined to produce multiple bundles, e.g., `mvn -Passet-cli,asset-protege` to generate both command line tool and Protégé bundles (from version 4.1 up to version 5 excluded, it was possible to build the RDF4 workbench + Ontop files by enabling profile `webapps`, now removed).

Compiling the Ontop Protégé plugin

Starting from version `5.0.2`, it is necessary to enable profile `plugin` (or a profile depending on it, such as `asset-plugin`, `asset-protege`, or `release`) in order to compile `ontop-`

`protege-plugin`. Note that commands `mvn compile` and `mvn test` will not work with profile `plugin` (or a depending profile) since compiling/testing `ontop-protege-plugin` requires building `ontop-protege-dependencies:shaded` first to [relocate Guava via the Maven shade plugin](#), but this happens only during the `package` lifecycle phase of Maven, i.e., starting from `mvn package` or later phase. Unfortunately, relocating Guava is needed to avoid conflicts with the older Guava version used in Protégé and cannot be avoided, and doing that before the `package` phase is not feasible. Therefore, developers should always use `mvn package -Pplugin` or later phase when working with `ontop-protege-plugin`, even when just compiling or testing is intended.

Other useful Maven commands

The following Maven commands can be used to facilitate some build-related development tasks, the involved plugins (e.g., `license-maven-plugin`) being already configured in Ontop root `pom.xml` file:

- `mvn license:add-third-party` - produces a report file `build/distribution/THIRD-PARTY.txt` useful to verify the use of Java libraries with compatible licenses
- `mvn org.codehaus.mojo:versions-maven-plugin:display-plugin-updates` - check for Maven plugins that can be updated
- `mvn org.codehaus.mojo:versions-maven-plugin:display-dependency-updates` - check for dependencies that can be updated
- `mvn dependency:analyze-duplicate` - check there are no duplicate dependencies declared in `pom.xml` files
- `mvn dependency:analyze-dep-mgt` - check that dependency versions under centralized `<dependencyManagement>` sections are not overridden in child `pom.xml` files
- `mvn dependency:analyze` - check for *used and undeclared* dependencies and for *unused and declared* dependencies (both situations that should be avoided)
- `mvn com.github.ekryd.sortpom:sortpom-maven-`

`plugin:sort` - sorts section of `pom.xml` files, to improve consistency

- `mvn net.revelc.code.formatter:formatter-maven-plugin:format` - auto-formats the `pom.xml` files, to improve consistency

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Git workflow](#)

[Make a new release →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
 - Update documents
 - Create a git branch for release
 - Update the version numbers
 - Build Maven packages
 - Build and distribute Ontop bundles
 - Update Docker Hub
 - Update the Ontop Website
 - Update the API example repository
 - Update branches for next development iteration
 - Time for celebration!
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Make a new release

Update documents

- `README.md` in `build/distribution`. This file will be uploaded to SourceForge and Github Releases
- `README.html` in `protege/plugin`. This file will be displayed in the Protégé Plugin Update window.
- `update.properties` file in `protege/plugin`. This is the configuration file for Protégé plugin Auto Update

Create a git branch for release

```
$ git checkout <main development branch>
$ git checkout -b releasing/<version-major.minor>
```

Update the version numbers

Update pom.xml files by Maven plugin to a SNAPSHOT version:

```
$ ./mvnw versions:set -DnewVersion=<version-major.minor>-SNAPSHOT
# $ ./mvnw versions:set -DnewVersion=5.0.0-SNAPSHOT
```

Build Maven packages

```
$ ./mvnw release:clean

# Preparing the release will sign artifacts (GPG key needed), create the new tag in git, and automatically push to github.
```

```
$ ./mvnw release:prepare -B

# stage the release
$ ./mvnw release:perform

# To skip tests:
# ./mvnw release:perform \
#   -Darguments="-Dmaven.test.skip=true -DskipTests" \
#   -Dgoals="deploy -Dmaven.test.skip=true -DskipTests"

# Or stage from a Git tag
# ./mvnw release:perform -DconnectionUrl=scm:git:git@github.com:ontop/ontop.git -Dtag=ontop-5.0.0
```

Stage the Maven artifacts at Maven Central

- URL: <https://central.sonatype.com/publishing/deployments>
- Click **Publish**

Build and distribute Ontop bundles

The bundles are located at

- CLI: `target/checkout/build/distribution/target/`
 - `ontop-cli-*.zip`
- Protege plugin and bundles:
 - `target/checkout/protege/distribution/target/`
 - `it.unibz.inf.ontop.protege-*.jar`
 - `ontop-protege-bundle-linux-*.tar.gz`
 - `ontop-protege-bundle-macos-*.zip`
 - `ontop-protege-bundle-win-*.zip`
 - `ontop-protege-bundle-platform-independent-*.zip`

We can also recompile them from the `target/checkout` directory generated during the Maven release

```
$ cd target/checkout
```



```
$ ./build-release.sh
```

Further info: [Build Instruction](#)

- Make a release in [Github Releases](#)
- Deploy the packages to [SourceForge](#) by manual upload or sftp.

```
$ sftp user,ontop4obda@frs.sourceforge.net
$ cd /home/pfs/project/o/on/ontop4obda
#
sftp://ghxiao,ontop4obda@frs.sourceforge.net/home/pfs/
```

Update Docker Hub

- Build and push the docker image of [ontop/ontop](#) to Docker Hub using the script `client/docker/build-image.sh`
 - by default, the script tag the images as `ontop/ontop:[version]` and `ontop/ontop:latest` - use option `-t` (one or more times) to tag differently (e.g., `-t ontop/ontop:xyz -t ontop/ontop:latest`)
 - run `client/docker/build-image.sh --help` for available options

```
# log in to Docker Hub
# docker login
$ cd ontop/client/docker/
# `-C`: multi-stage build. slow, but safe
# `-x`: compile for both amd64 and arm64, which
# entails pushing to Docker hub
$ ./build-image.sh -C -x
```

Update the Ontop Website

- Update [the guide homepage](#)
- Update [the release notes](#)

Update the API example repository

- Update the Ontop version in <https://github.com/ontop/ontop-api-examples>

Update branches for next development iteration

- Merge the tag for the release to master and the release branch to develop

```
$ git checkout develop
$ git merge release/v-number #branch

# this step is also for triggering the AutoUpdate
of Protege plugin
$ git checkout master
$ git merge v-number # tag
```

Time for celebration!

- Send release emails to several mail lists and social media

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Build Ontop Bundles](#)

[Build snapshots →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Maven
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Build snapshots

Maven

Add the credentials for `central` (Maven Central) in your `.m2/settings.xml`.

```
mvn deploy -DskipTests -pl '!protege/dependencies,!protege/plugin,!protege/distribution' -s .
```

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← Make a new release

Debug Protégé →

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
 - Build the Protégé plugin
 - Debug in IntelliJ using the Remote debugger
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Debug Protégé

This page describes how to debug the Protégé Plugin of Ontop from IntelliJ.

If you just want to test the plugin (without debugging), you only need to follow the step of building the Protégé plugin, and then run Protégé as usual (use `run.bat` on Windows)

```
$ Protege-[protege-version]/run.sh
```

Build the Protégé plugin

Requirements

Checkout and build Ontop source code as described [here](#). We assume all commands listed below are issued from the root directory of Ontop source tree.

First time build

Unzip the Ontop Protégé bundle (obtained by previously building Ontop source code), which contains Protégé preconfigured with the Ontop Protégé plugin

```
$ unzip protege/distribution/target/ontop-protege-bundle-platform-independent-[ontop-version].zip
```

Notes:

- for older versions of Ontop (before 5.0.0) the bundle to unzip is located either in `build/distribution/ontop-protege` (up to 4.0.3) or in `build/distribution/target` (up to 5.0.0 excluded);

- you may also reuse a (platform independent) Protégé distribution obtained directly from Protégé web site and already installed locally. In this case make sure that the Protégé version matches the one used during Ontop build, that Protégé directory contains a `plugins` sub-directory (create it if missing), and run the commands below to deploy the Ontop Protégé plugin (adapting paths accordingly).

Subsequent builds

Each time you modify the code, you need to recompile and build the Ontop distribution.

The fastest command to re-compile the Ontop Protégé plugin and the code it depends on is

```
$ mvn clean package -Dmaven.test.skip -Passet-plugin -pl protege/distribution -am
```

Where:

- `clean package` forces a full re-build and may be generally omitted (which defaults to running `package`) unless you encounter build errors, in which case `clean` helps ruling out they stem from incremental builds;
- `-Dmaven.test.skip` disables compiling and executing unit tests;
- `-Passet-plugin` activates the generation of the Ontop Protégé plugin (OSGI bundle with all its dependencies), without including other unneeded artifacts (e.g., full ZIP/tar.gz Protégé bundles, source and Javadoc JARs, etc);
- `-pl protege/distribution -am` (optional) tells Maven to only build the Ontop Protégé module located in `protege/distribution` and all the other Ontop modules it depends on, saving another couple of seconds.

Notes:

- in Ontop versions 4.1 to 5.0.0 excluded, the command above should be replaced by `mvn clean package -`

```
mvn test -Dmaven.test.skip -Pprotege -pl client/protege -am ;
```

- in Ontop version 4.0.3 or earlier, the command above should be replaced by `mvn clean install -pl client/protege -am -DskipTests` to compile code and install it in local maven repository, followed by `mvn bundle:bundle -pl client/protege` to package the plugin.

Once the Ontop Protégé plugin is built, it must be copied to the Protégé plugin directory via

```
$ cp
protege/distribution/target/it.unibz.inf.ontop.protege
[ontop-version].jar Protege-[protege-
version]/plugins/
```

The command above should replace any previous Ontop Protégé plugin in the `plugins` directory. In any case, make sure that there is only one Protégé plugin in `Protege-[protege-version]/plugins/`

Debug in IntelliJ using the Remote debugger

IntelliJ configuration

Create a remote configuration:

- select 'Edit configurations', click '+', then select 'Remote'
- Host: `localhost`
- Port: choose a port number (`5005` in what follows)

Protégé run options

Edit the script `Protege-[protege-version]/run.sh` (on Windows: `run.bat`), adding the following JVM option (note that parameter `suspend` is set to `y`)

```
-
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,a
```

Run / Debug

Execute `Protege-[protege-version]/run.sh` (on Windows: `run.bat`). Then, from IntelliJ, click on the Debug button (or Shift+F9).

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Build snapshots](#)

[Debug Jetty →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty**
 - Start the Jetty in Debug mode
 - Linking with IntelliJ
 - References
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Debug Jetty

This page describe debug Jetty with RDF4J workbench plugin in IntelliJ Idea

Start the Jetty in Debug mode

```
$ java -Xdebug -
agentlib:jdwp=transport=dt_socket,address=9999,server=
-jar start.jar
```

Linking with IntelliJ

- Next we need to link the IntelliJ project with the deployed webapp.
1. Within IntelliJ, open the project containing the webapp deployed into jetty that you want to debug. SelectRun -> Edit Configurations. Add a new configuration by clicking the "+" icon. Choose Remote. Make sure the port you choose is the same as the one you added in Enable remote debugging.
 2. Next in your webapp you can set a breakpoint within a servlet which when it is tripped will halt the remote jvm's processing thread to await for debugging commands from your IntelliJ instance. To set a breakpoint, simply open the servlet or any other class you want to debug and click left to the line you want to set the breakpoint at (where the red dot is on the next screenshot). The red dot and red background on the line mark the breakpoint.
 3. Accessing that servlet within your browser, pointed at your remote debug configured jetty-distribution, should transition

your IntelliJ instance to the standard debugger view.

References

- 1. <http://www.eclipse.org/jetty/documentation/current/enable-remote-debugging.html>
- 2. <http://www.eclipse.org/jetty/documentation/current/debugging-with-intellij.html>

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← [Debug Protégé](#)

[Support a new database system](#) →

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
 - Required implementations
 - Optional implementations
 - Declaring an implementation
 - Preparing and Running the Testcases
 - Further Implementations and Troubleshooting
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Support a new database system

This page describes how Ontop can be extended to support a novel relational database system (RDBMS).

The following RDBMSs are currently supported:

- DB2
- MySQL / MariaDB
- Oracle
- PostgreSQL
- SQL server
- H2
- Denodo
- Dremio
- Apache Spark SQL & Databricks
- Teiid
- Snowflake
- Trino
- Presto
- Athena
- Redshift
- DuckDB

It is relatively easy though to extend Ontop's source code in order to support an additional RDBMS, thanks to dependency injection.

New: Guided Implementation

Ontop also provides a [python tool](#) that automatically generates a large part of the required code end simplifies the process of developing support for new SQL dialects.

Using it minimizes the overhead for the implementation

of new dialect support and allows you to test your code easily right away.

This page provides a detailed explanation of the key concepts that need to be implemented, and was updated with Ontop 5.0.2. For newer versions of Ontop, it may no longer be up-to-date.

Required implementations

Two implementations must be provided, each for a different interface that already has a default (abstract) implementation.

These implementations dictate what the datatypes of the RDBMS are and how certain function symbols can be translated into the SQL dialect supported by the RDBMS.

The default implementation of each of these interfaces is often sufficient to handle many query operators and functions. As a result, only a few methods generally need to be overwritten, to account for specificities of the new SQL dialect.

The 2 required implementations are the DB function symbol factory and the DB datatype factory.

DB function symbol factory

The *DB function symbol factory* provides function symbols which can be directly serialized into the target SQL dialect (e.g. `LENGTH` or `CURRENT_TIMESTAMP`). This factory has methods for common operations with precise semantics (e.g. concatenating 3 string arguments in a null-rejecting manner).

The interface to implement is `DBFunctionSymbolFactory`. And the default implementation is `AbstractSQLDBFunctionSymbolFactory`.

For instance, the DB function symbol factory for PostgreSQL within Ontop is the class `PostgreSQLDBFunctionSymbolFactory`.

WARNING

This interface should not be confused with the `FunctionSymbolFactory`, which is in charge of constructing SPARQL function symbols and other function symbols that cannot be directly serialized into SQL.

A basic template for this implementation is provided here:

```

java
public class NewDialectDBFunctionSymbolFactory
    extends AbstractSQLDBFunctionSymbolFactory {

    @Inject
    protected
    DuckDBDBFunctionSymbolFactory(TypeFactory
    typeFactory) {

        super(createNewDialectRegularFunctionTable(typeFactory
        typeFactory));
    }

    protected static ImmutableTable<String,
    Integer, DBFunctionSymbol>
    createNewDialectRegularFunctionTable(TypeFactory
    typeFactory) {
        DBTypeFactory dbTypeFactory =
        typeFactory.getDBTypeFactory();
        DBTermType abstractRootDBType =
        dbTypeFactory.getAbstractRootDBType();

        Table<String, Integer, DBFunctionSymbol>
        table = HashBasedTable.create(

        createDefaultRegularFunctionTable(typeFactory));

        /*
        -----
        -
        -           Changed Function Symbols
        -
        -----
        */
    }

```

```

        return ImmutableTable.copyOf(table);
    }

    /*
     -----
     -               Implementations             -
     -----
    */

}

```

In the template above, two code blocks are commented out. We will now look at each of them individually.

Changed Function Symbols

The *regular function table* is a dictionary, that maps from function names to the function symbol instance it corresponds to. The abstract base classes we inherit from already define a default function symbol table that includes all commonly used SQL functions. In some cases, however, some of these default function symbols need to be changed. For these cases, new entries can be created in this block, to override existing, bad entries.

A popular example is the `CURRENT_TIMESTAMP` function symbol. By default, it is created as a simple function with arity 0 of the form `CURRENT_TIMESTAMP()`. In some dialects, however, the current time is called without parentheses, simply in the form `CURRENT_TIMESTAMP`. In this case, we need to override the function symbol created by the default implementation, to not include parentheses:

```

                                                                    java
DBFunctionSymbol nowFunctionSymbol = new
WithoutParenthesesSimpleTypedDBFunctionSymbolImpl(
    CURRENT_TIMESTAMP_STR,
    dbTypeFactory.getDBDateTimestampType(),
    abstractRootDBType);
table.put(CURRENT_TIMESTAMP_STR, 0,
nowFunctionSymbol);

```

The `CURRENT_TIMESTAMP` function is now a `WithoutParenthesesSimpleTypedDBFunctionSymbol`, so when generating SQL queries, ontop will not include the parentheses. By putting this new function symbol into the table at the key `CURRENT_TIMESTAMP_STR`, we override the default behaviour that was put into the table by the default method.

Implementations

A set of serialization methods have been left abstract by the base classes. These *always* have to be implemented for each new dialect.

All such methods share the same structure:

```

protected String
serialize[FUNCTION_NAME](ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    ...
}
    
```

Here, `terms` is the list of arguments that were passed to the function, `termConverter` is a function that can be used to transform any given term into its SQL equivalent, and `termFactory` can be used to generate new terms that may be required for some functions.

We will now go through each of them and provide example implementations.

Contains:

This function returns a boolean value, indicating whether a given string (argument 2) is included in a different string (argument 1). While some dialects have their own `CONTAINS` functions, for others it may have to be implemented through a string search, like in this example:

```

@Override
protected String serializeContains(ImmutableList<?
    
```

```
extends ImmutableTerm> terms,
Function<ImmutableTerm, String> termConverter,
TermFactory termFactory) {
    return String.format("(POSITION(%s IN %s) >
0)",

termConverter.apply(terms.get(1)),

termConverter.apply(terms.get(0)));
}
```

StrBefore:

This function returns the section of a given string (argument 1) before the first appearance of a given search string (argument 2). If the searched string is not contained in the base string, an empty string is returned instead. In many languages, this can be implemented using string search and substring methods.

```
@Override
protected String
serializeStrBefore(ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    String before =
termConverter.apply(terms.get(1));

    return
String.format("SUBSTRING(%s,1,POSITION(%s IN
%s)-1)", str, before, str);
}
```

StrAfter:

The opposite of `STRBEFORE`, this function returns the section of a given string (argument 1) *after* the first appearance of a search string (argument 2). This function can be implemented similarly:

```
@Override
protected String
serializeStrBefore(ImmutableList<? extends
```

```
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    String before =
termConverter.apply(terms.get(1));

    return String.format("SUBSTRING(%s,
IF(POSITION(%s IN %s) != 0, POSITION(%s IN %s) +
LENGTH(%s), 0))", str, after, str, after, str,
after);
}
```

Tz: This function extracts the timezone part of a given timestamp. An example implementation may look like this:

java

```
@Override
protected String serializeTz(ImmutableList<?
extends ImmutableTerm> terms,
Function<ImmutableTerm, String> termConverter,
TermFactory termFactory) {
    String str =
termConverter.apply(terms.get(0));
    return String.format("
(LPAD(EXTRACT(TIMEZONE_HOUR FROM %s)::text,2,'0')
|| ':' || LPAD(EXTRACT(TIMEZONE_MINUTE FROM
%s)::text,2,'0'))", str, str);
}
```

DateTimeNorm: This function converts a given timestamp to the standardized `ISO 8601` format. An example implementation may look like this:

java

```
@Override
protected String
serializeDateTimeNorm(ImmutableList<? extends
ImmutableTerm> terms, Function<ImmutableTerm,
String> termConverter, TermFactory termFactory) {
    return String.format("TO_IS08601(%s)",
termConverter.apply(terms.get(0)));
}
```


Hashing Functions:

Ontop supports a set of hasing functions:

- MD5
- SHA1
- SHA256
- SHA384
- SHA512

For each of them, the implementation of a serialization method is required. An example implementation of the MD5 function may look like this:

```
java
@Override
protected String serializeMD5(ImmutableList<?
    extends ImmutableTerm> terms,
    Function<ImmutableTerm, String> termConverter,
    TermFactory termFactory) {
    return String.format("MD5(%s)",
        termConverter.apply(terms.get(0)));
}
```

Other than the serialization methods, four additional methods must be implemented by the subclass:

```
java
@Override
protected String getUUIDNameInDialect() { return "
    [UUID_FUNCTION_NAME]"; }
```

returns the string that represents the name of the UUID in the SQL dialect.

The methods `createNullRejectingDBConcat(int arity)`, `createDBConcatOperator(int arity)`, and `createRegularDBConcat(int arity)` need to be implemented to define the concat function symbols. The `DBConcat` methods define function symbols that represent `CONCAT` function calls of a given arity. Sample implementations for them may look like this:

```
java
```

```
@Override
protected DBConcatFunctionSymbol
createRegularDBConcat(int arity) {
    return new
    NullToleratingDBConcatFunctionSymbol("CONCAT",
    arity, dbStringType, abstractRootDBType, false);
}

@Override
protected DBConcatFunctionSymbol
createNullRejectingDBConcat(int arity) {
    return new
    NullRejectingDBConcatFunctionSymbol("CONCAT_NO_NULL",
    arity, dbStringType, abstractRootDBType, false);
}
```

The `createDBConcatOperator(int arity)` method represents a concat operation executed through a concat operator (often `+` or `||`). It may be implemented like this:

```
@Override
protected DBConcatFunctionSymbol
createDBConcatOperator(int arity) {
    return new
    NullRejectingDBConcatFunctionSymbol("||", arity,
    dbStringType, abstractRootDBType,
    Serializers.getOperatorSerializer("||"));
}
```

java

DB Datatype factory

The *DB datatype factory* declares the hierarchy of DB datatypes used by the DBMS, and specifies their correspondence with datatypes used in the RDF graphs (such as xsd datatypes).

The interface to implement (for SQL dialects) is

```
SQLDBTypeFactory
```

And the default implementation is `DefaultSQLDBTypeFactory`.

For instance, the datatype factory for PostgreSQL within Ontop is the class `PostgreSQLDBTypeFactory`.

A basic template for this implementation is provided here:

java

```
public class NewDialectDBTypeFactory extends
DefaultSQLDBTypeFactory {

    /*
        -----
        -           Type Definitions           -
        -----
    */

    @AssistedInject
    protected NewDialectDBTypeFactory(@Assisted
TermType rootTermType, @Assisted TypeFactory
typeFactory) {

super(createNewDialectTypeMap(rootTermType,
typeFactory), createNewDialectCodeMap());
    }

    protected static Map<String, DBTermType>
createNewDialectTypeMap(TermType rootTermType,
TypeFactory typeFactory) {
        /*
            -----
            -           Type Map           -
            -----
        */
    }

    protected static ImmutableMap<DefaultTypeCode,
String> createNewDialectCodeMap() {
        /*
            -----
            -           Code Map           -
            -----
        */
    }

    /*
```

```
-----  
-               Support Flags               -  
-----  
  
    */  
  
}
```

In the template above, four code blocks are commented out. We will now look at each of them individually.

Type Definitions

In this section, we define a set of `String` constants that represent the names of datatypes in our SQL dialect. These constants are then used in other code blocks. An excerpt of the *type definitions* part in an implementation may look like this:

```
protected static final String BIT_STR = "BIT";  
protected static final String INT2_STR = "INT2";  
protected static final String INT4_STR = "INT4";  
protected static final String INT8_STR = "INT8";  
protected static final String FLOAT4_STR =  
    "FLOAT4";  
protected static final String FLOAT8_STR =  
    "FLOAT8";
```

TYPE MAP

In this method, we define the contents of the type map for our dialect. We create `DBTermType` instances for each supported type and store them in a `Map` object, that matches their identifiers to the `DBTermType`. The base class we extend from already generates an extensive type map for many popular types, such as `VARCHAR`, `INTEGER`, `REAL`, `DOUBLE`, and more in the method `createDefaultSQLTypeMap(...)`. New types can be added to the map in the subclass, and, alternatively, we can override existing entries in the map here. An example implementation of this method may look like this:

```
protected static Map<String, DBTermType>
```

```
createNewDialectTypeMap(TermType rootTermType,
TypeFactory typeFactory) {
    TermTypeAncestry rootAncestry =
rootTermType.getAncestry();

    Map<String, DBTermType> map =
createDefaultSQLTypeMap(rootTermType,
typeFactory);

    return map;
}
```

CODE MAP

The code map is a dictionary, that maps different type codes to their "default type" for the SQL dialect. This includes the following type codes:

- `STRING`
- `HEXBINARY`
- `LARGE_INTEGER`
- `DECIMAL`
- `DOUBLE`
- `BOOLEAN`
- `DATE`
- `TIME`
- `DATETIMESTAMP`
- `GEOMETRY`
- `GEOGRAPHY`
- `ARRAY`
- `JSON`

The default implementation already maps many of these to the name of the corresponding data type. To support additional ones, or to override existing mappings, this method can be implemented. An example implementation may look like this:

```
protected static ImmutableMap<DefaultTypeCode,
String> createNewDialectCodeMap() {
    Map<DefaultTypeCode, String> map =
createDefaultSQLCodeMap();
}
```

java

```
map.put(DefaultTypeCode.JSON, JSONB_STR);

return ImmutableMap.copyOf(map);
}
```

Support Flags

An instance of `DBTypeFactory` supports a set of methods that, when called, return a flag indicating if a certain SQL feature is supported by the dialect. These methods are

- `supportsDBGeometryType()`
- `supportsDBGeographyType()`
- `supportsDBDistanceSphere()`
- `supportsJson()`
- `supportsArrayType()`

Any of these functions can be overridden by the `DBTypeFactory` sub-class to return a boolean constant indicating if the type is supported.

Other Implementations

The `DBTypeFactory` base-class supports further methods that can be included in sub-classes for fine-tuning. One popular set of such methods are the `LexicalValue` methods. These return a `String` constant that represents the lexical value the corresponding literal may assume. For instance, by default, the method `getDBFalseLexicalValue()` returns the string `"FALSE"`, as this is a common representation of the boolean `false` value in many SQL dialects. To implement the `DBTypeFactory` for a dialect that instead uses the digit `0` for `false` literals, we could override the method like so:

```
@Override
public String getDBFalseLexicalValue() {
    return "0";
}
```

java

Optional implementations

Additional implementations can be optionally provided in replacement of the default implementation.

Serializer

The *serializer* dictates the general shape (SELECT-FROM-WHERE) of the SQL query.

The interface to implement is `SelectFromWhereSerializer`.
And the default implementation is `DefaultSelectFromWhereSerializer`.

For instance, the serializer for PostgreSQL within Ontop is the class `PostgresSelectFromWhereSerializer`.

Normalizer

The *normalizer* addresses limitations of certain DBMSs (such as a non-canonical evaluation of the ORDER BY clause).

The interface to implement is `DialectExtraNormalizer`.
And several implementations are already available, some of which are used by several DBMS.

For instance, the normalizer associated to PostgreSQL within Ontop is

`OnlyInPresenceOfDistinctProjectOrderByTermsNormalizer`.

Metadata provider

The *metadata provider* specifies how schema and integrity constraints (for instance primary keys) are retrieved from the DBMS.

The interface to implement is `MetadataProvider`.
And the default implementation is `DefaultDBMetadataProvider`.

For instance, the metadata provider for PostgreSQL within Ontop is `PostgreSQLDBMetadataProvider`.

Declaring an implementation

All the implementations mentioned above can be declared in the property file `sql-default.properties` (which can be found in the directory `it/unibz/inf/ontop/injection/` of the `ontop-rdb` module).

A key-value pair must be added for each of these implementations, where the key indicates the type of the implementation (serializer, function symbol factory, etc.), and the value is the implementation.

The naming scheme for the keys is the following.

Let `<driverName>` be the name of the JDBC driver for the RDBMS (for instance, the JDBC driver for PostgreSQL is `org.postgresql.Driver`).

Then the keys are:

- `<driverName>-serializer` for a serializer
- `<driverName>-symbolFactory` for a DB function symbol factory
- `<driverName>-typeFactory` for a DB datatype factory
- `<driverName>-normalizer` for a normalizer
- `<driverName>-metadataProvider` for a metadata provider

For instance, the key-value pairs declared for PostgreSQL are:

```
org.postgresql.Driver-serializer =
it.unibz.inf.ontop.generation.serializer.impl.Postgres

org.postgresql.Driver-symbolFactory =
it.unibz.inf.ontop.model.term.functionsymbol.db.impl.P

org.postgresql.Driver-typeFactory =
it.unibz.inf.ontop.model.type.impl.PostgreSQLDBTypeFac

org.postgresql.Driver-normalizer =
it.unibz.inf.ontop.generation.normalization.impl.OnlyI
```



```
org.postgresql.Driver-metadataProvider =
it.unibz.inf.ontop.dbschema.impl.PostgreSQLDBMetadataP
```

A basic set of key-value pairs may be:

```
name.of.jdbc.Driver-serializer =
it.unibz.inf.ontop.generation.serializer.impl.DefaultS

name.of.jdbc.Driver-symbolFactory =
it.unibz.inf.ontop.model.term.functionsymbol.db.impl.N

name.of.jdbc.Driver-typeFactory =
it.unibz.inf.ontop.model.type.impl.NewDialectDBTypeFac

name.of.jdbc.Driver-normalizer =
it.unibz.inf.ontop.generation.normalization.impl.OnlyI

name.of.jdbc.Driver-metadataProvider =
it.unibz.inf.ontop.dbschema.impl.DefaultSchemaCatalogD
```

Preparing and Running the Testcases

Ontop currently uses a set of tests called `lightweight-tests` to test all main ontop language support features on minimal databases. A simple way of testing the correctness of newly implemented dialect support is to run the same tests a database that uses the corresponding dialect.

This consists of the following steps:

1. Set up a database with the required testing conditions
2. Prepare files that assist the connection to the database
3. Implement sub-classes of the abstract lightweight test classes and run them

In the following sections, we will discuss each of these steps.

Setting up the Database

The exact procedure of the database preparation depends on the database system. In ontop, we generally use docker images that reproduce the exact same database state for each test run so we can test the different dialects automatically.

Each directory inside `test/lightweight-tests/lightweight-db-test-images/` corresponds to one docker image, and many of them contain table definitions in `.sql` file. For instance, the `mysql` directory contains the exact definitions of the three databases that are used for testing, `books`, `dbconstraints`, and `university`.

To prepare the required testing conditions, you can take these `.sql` files and run their queries on your database. Should any of the features not be supported, you will be required to either use alternative equivalent features to reach the same final database state, or disable the corresponding test cases.

Preparing Connection Files

The directory `test/lightweight-tests/src/test/resources/` contains files that are used for setting up an ontop connection with different databases, including `.obda` files for the VKG creation and `.property` files for JDBC connections. While the `.obda` files can usually be reused for different systems, the `.property` files must be included for each database system. The directories `books` and `prof` inside the test resources contain one sub directory for each dialect, where each sub directory in turn includes one `.properties` file for the JDBC connection. The `dbconstraints` and `university` directories, on the other hand, directly contain `.property` files for each dialect.

To run all lightweight tests for a new dialect, the corresponding files and directories have to be created for its JDBC connection. In particular, the connection files should be created in such a way, that the corresponding database/schema of the test is selected as the default database/schema, as the `.obda` files access the

tables through relative paths.

Implementing the Lightweight Test Classes

The directory `test/lightweight-tests/src/test/java/` contains the test cases that have to run successfully for each new dialect. They are defined in the four abstract classes

`AbstractBindTestWithFunctions`, `AbstractConstraintTest`, `AbstractDistinctInAggregateTest`, and `AbstractLeftJoinProfTest`.

For each new dialect, a new package has to be created at this location, containing the implementations of sub-classes for each of these abstract base classes.

`BindTestWithFunctions`

This test class tests all main SQL functions that are supported by ontop. It uses the `books` database, as well as directly generated data through the SPARQL `BIND` function. An example implementation of this class may look like this:

```
java
@NewDialectLightweightTest
public class BindWithFunctionsNewDialectTest
extends AbstractBindTestWithFunctions {

    private static final String PROPERTIES_FILE =
        "/books/new-dialect/books-new-dialect.properties";

    @BeforeAll
    public static void before() throws
        IOException, SQLException {
        initOBDA(OBDA_FILE, OWL_FILE,
            PROPERTIES_FILE);
    }

    @AfterAll
    public static void after() throws SQLException
    {
        release();
    }
}
```

```

    /*
       -----
       -               Result Definitions               -
       -----
    */

}

```

NOTE

Using the `@NewDialectLightweightTest` annotation requires the definition of the annotation in the same package, similarly to how it was defined for other dialects.

The code block "Result Definitions" is commented out in the above example. This code block may contain redefinitions of expected results that are tested in the base class. To do this, override the `get[TEST-NAME]ExpectedValues()` method here. An example implementation may look like this:

```

@Override
protected ImmutableList<String>
getConstantIntegerDivideExpectedResults() {
    return
    ImmutableList.of("\0.500\^^xsd:decimal");
}

```

Should any of the functionalities not be supported by the SQL dialect, then the corresponding test case may simply be disabled with an explanation as to why it is not supported. An example for that may look like this:

```

@Disabled("New Dialect does not support SHA384")
@Test
@Override
public void testHashSHA384() {
    super.testHashSHA384();
}

```

Some test cases are disabled by default.

Constraint

This set of test cases tests for the correct interpretation of integrity constraints in the data. It uses the `dbconstraints` database. An example implementation may look like this:

```

@NewDialectLightweightTest
public class ConstraintNewDialectTest extends
AbstractConstraintTest {

    private static final String PROPERTIES_FILE =
"/dbconstraints/dbconstraints-new-
dialect.properties";

    public ConstraintNewDialectTest(String method)
    {
        super(method, PROPERTIES_FILE);
    }

}
    
```

Once again, specific tests may be disabled if they are not supported. For database systems that do not support integrity constraints, this class can be ignored.

DistinctInAggregate

This set of test cases runs queries containing aggregate functions that include the `DISTINCT` keyword. It can be implemented analogously to the `BindWithFunctions` test class.

LeftJoinProf

This set of test cases runs on the `university` database. It accesses items that have to be taken from the database through `JOIN`s. Furthermore, it tests the optimization procedures employed by ontop to minimize the required number of joins. Once again, it can be implemented analogously to the `BindWithFunctions` test class.

If the database management system does not support integrity constraints, many of these test cases will fail because the optimizations will not be performed.

Further Implementations and Troubleshooting

Once the basic features and lightweight test cases have been implemented for a dialect, the tests can be run to determine which features still do not work correctly.

If a test case fails, its corresponding feature is likely not yet supported by the current implementation. In these cases, changes need to be applied to the implemented `DBTypeFactory` or `DBFunctionSymbolFactory` to solve the issue.

In some situations, however, issues may be caused by problems not related to database types or functions. In these cases, one of the following default implementations may also be extended for the new dialect.

NOTE

New implementations of these classes have to be linked to the driver through the `sql-default.properties` file.

```
DefaultSchemaCatalogDBMetadataProvider,  
DefaultSchemaDBMetadataProvider, or  
DefaultDBMetadataProvider
```

If the `DBMetadataProvider` assigned to the dialect is not able to perform its expected actions through its default implementation, it may have to be extended with a new sub-class that is particularly tailored for this dialect.

In that case, its methods may be overridden to change the default behaviour. Common targets include:

`insertIntegrityConstraints(...)` if the dialect does not support integrity constraints.

- `getRelationId(...)` to return the ID of a given relation.

Additionally, the constructor may be implemented to determine the default schema or default catalog names if they cannot be accessed directly through the JDBC, or a different

`QuotedIDFactory` instance can be passed to the super-class to force ontop to generate quoted identifiers in a specific way.

SelectFromWhereSerializer

This class is responsible for the serialization of the SQL query frame. Different SQL dialects have different rules as to how the general shape of an SQL query have to be defined. For instance, some dialects allow users to combine `LIMIT` and `OFFSET` clauses as `LIMIT <offset>,<limit>`, while others require the form `OFFSET <offset> LIMIT <limit>`. The following shows a default implementation of a `SelectFromWhereSerializer` to achieve the latter behaviour:

```
java
@Singleton
public class NewDialectSelectFromWhereSerializer
extends DefaultSelectFromWhereSerializer
implements SelectFromWhereSerializer {

    @Inject
    private
    NewDialectSelectFromWhereSerializer(TermFactory
termFactory) {
        super(new
DefaultSQLTermSerializer(termFactory));
    }

    @Override
    public QuerySerialization
serialize(SelectFromWhereWithModifiers
selectFromWhere, DBParameters dbParameters) {
        return selectFromWhere.acceptVisitor(
            new
DefaultRelationVisitingSerializer(dbParameters.getQuot
{
```

```
        @Override
        protected String
serializeLimitOffset(long limit, long offset,
boolean noSortCondition) {
            return
String.format("OFFSET %d LIMIT %d", offset,
limit);
        }
    });
}
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Debug Jetty](#)

[Statistics →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system

Statistics

Downloads

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Statistics

Downloads

Github releases

For all the versions 4.x:

```
curl -H "Accept: application/vnd.github.v3+json" sh
https://api.github.com/repos/ontop/ontop/releases
\
  | jq '[] | select((.tag_name |
startswith("ontop-4."))) | .assets[] |
.download_count ] | add'
```

For all the versions 5.x:

```
curl -H "Accept: application/vnd.github.v3+json" sh
https://api.github.com/repos/ontop/ontop/releases
\
  | jq '[] | select((.tag_name |
startswith("ontop-5."))) | .assets[] |
.download_count ] | add'
```

Docker Hub (image pulls)

Starting from 5.0.0:

```
curl -s sh
"https://hub.docker.com/v2/repositories/ontop/ontop/"
| jq '.pull_count'
```

Before:

```
curl -s
"https://hub.docker.com/v2/repositories/ontop/ontop-
endpoint/" | jq '.pull_count'
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Support a new database system](#) [Intermediate Query \(IQ\)](#) [→](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- IQ
- Query nodes
- Previous query format (version 1)
- Guice Conventions

Outdated corner ▶

Intermediate Query (IQ)

Input queries for Ontop are expressed in (a fragment of) SPARQL 1.1, whereas the output of the query reformulation process is expressed in the query language used by the underlying DBMS (prototypically some variant of SQL but also other query languages like the MongoDB Aggregate Framework).

Because most query rewriting and optimization steps apply regardless of the underlying DBMS, a DBMS-independent query representation format is used internally. This format is called *intermediate query* (IQ) and allows for the representation of queries and mapping assertions.

IQ

In terms of implementation, since 4.3.0, intermediate queries are always immutable and based on the class `IQ`. Prior to that, there was also a mutable variant, centered on the former class `IntermediateQuery`, which was actually the first implementation.

Query nodes

An IQ is a standard (rooted and ordered) tree representation of a algebraic query expression. In terms of implementation, it is based on a projection atom and an `IQTree`.

The underlying algebra is a compromise between (a fragment of) the SPARQL algebra on the one hand, and the select/project/join/rename/union relational algebra (RA in what follows) with named attributes on the other hand.

The following is an informal description of the different types of nodes which may appear in an IQ. A formal characterization (in

terms of RA algebra) is given in [the dedicated section](#).

Optimizations applied to IQs are described in [the dedicated section](#).

Leaf node types

Intensional data node

A intensional node is a prototypical leaf node that is expected to be replaced by an `IQTree` when unfolding the mapping.

It is characterized by a predicate P , where each x_i is either a ground term (constant or functional term without variable) or a variable name.

Intensional data nodes are typically used for representing triple and quad patterns from SPARQL.

Extensional data node

An extensional data node is similar to an intensional data node, expect that:

- It is not required to be unfolded (e.g. if it represents a database table)
- It has a sparse representation: only a subset of columns are mapped to variables or ground terms.

Empty node

An empty node can be viewed as a specific type of data node, whose evaluation will always be an empty set of tuples.

See [the dedicated section](#) for a precise characterization.

True node

True nodes represent another limit case, namely an expression whose evaluation will always be a singleton set containing the 0-ary tuple.

See [the dedicated section](#) for a precise characterization.

Native node

A native node is defined from a native query string (e.g. a SQL string) and explicits the datatypes of the variables returned by the native query. It is generated at the end of query reformulation. It is not used in the mapping.

Values node

Since 4.2.0. A Values node, similarly to the `VALUES` keyword in SQL and SPARQL, represents an "embedded table" having its rows embedded in the query.

Non-leaf node types

Filter node

A filter node represents an RA selection over the evaluation of its (unique) child node.

Inner join node

An inner join node represents the RA n-ary natural join of the evaluations of its children nodes.

Explicit joining conditions may also be attached to it.

See [the dedicated section](#) for a precise characterization.

Left join node

A left join node represents the RA binary natural left outer join of the evaluations of its children nodes. The left and right children nodes respectively stand for the left and right part of the join.

Explicit joining conditions may also be attached to it.

See [the dedicated section](#) for a precise characterization.

Union node

A union node represents the RA n-ary union of the evaluations of its children nodes, possibly preceded by a (unique) projection applied to each of them.

See [the dedicated section](#) for a precise characterization.

Construction node

A construction node represents a sequence of (at most) three operations applied to the evaluation of its unique child node:

- a projection, followed by
- the application of a function to the values of each retrieved tuple, followed by
- a variable renaming.

For instance, a construction node may first project out variables and , then prepend the string "http://domain/" to each value retrieved for , and finally rename variable with .

Each of these three operations is optional.

See [the dedicated section](#) for a precise characterization.

Aggregation node

An aggregation node is defined by the grouping variables over which the aggregation occurs (if any) and a substitution defined the new variables obtained by from aggregation functions.

It projects the grouping variables and the variables defined by the substitution.

Slice node

A slice node contains a limit and/or an offset value, that is uses for eliminate tuples from the result set.

Distinct node

A distinct node keeps only one occurrence of each tuple.

Order by node

An order by node orders the tuples based on a list of comparators. It orders first the results according to the first comparator (ascending or descending) and, in case of same ordering, it orders them using the following comparator, and so on.

This operator follows a **NULLS FIRST** semantics (if ascending, **NULLS LAST** otherwise), similarly to SPARQL.

Previous query format (version 1)

Datalog was the main internal query representation format in Version 1 of Ontop, due in particular to its close similarity with conjunctive queries, which were the only supported input queries. The switch to SPARQL input queries increased expressiveness, making their rule-based representation more complex (due in particular to the the SPARQL OPTIONAL operator, which, roughly speaking, corresponds to the SQL left outer join). The intermediate query format on the other hand offers a relatively straightforward internal representation, not only of SPARQL queries, but also of the queries sent to underlying DBMS (at least the SQL-like ones), as well as complex mapping assertions (which, in most scenarios, contains SQL queries). This is also why the rule-based internal representation has been progressively abandoned.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Statistics](#)

[Guice Conventions →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner ▶

Guice Conventions

Each module contains a folder `src/main/java/` and a module `it.unibz.inf.ontop.injection.filename.properties` (e.g., `mapping-default.properties`).

The file specifies to which implementation each interface needs to be instantiated. For instance, `mapping-default.properties` contains the following lines

```
#####
# Default implementations
#####

it.unibz.inf.ontop.spec.OBDASpecificationExtractor
=
it.unibz.inf.ontop.spec.impl.DefaultOBDASpecificationE

it.unibz.inf.ontop.spec.mapping.transformer.MappingTra
=
it.unibz.inf.ontop.spec.mapping.transformer.impl.Defau

...

```

`DefaultOBDASpecificationExtractor` is an implementation for the interface `OBDASpecificationExtractor`. The constructor, private, is injected, that is it has the `@Inject` tag at its beginning.

```
@Inject
private
DefaultOBDASpecificationExtractor(MappingExtractor
mappingExtractor, MappingTransformer
mappingTransformer,

OntopMappingSettings settings) {
    this.mappingExtractor = mappingExtractor;
}

```



```

        this.mappingTransformer =
mappingTransformer;
    }
}

```

The `@Inject` means that the arguments to the constructor will be provided by the injection mechanism. For instance, the constructor for the second parameter is:

```

@Inject
    private
DefaultMappingTransformer(MappingVariableNameNormalize
mappingNormalizer,

MappingSaturator mappingSaturator,

ABoxFactIntoMappingConverter inserter,

MappingMerger mappingMerger,

OntopMappingSettings settings,

MappingSameAsInverseRewriter
sameAsInverseRewriter,

SpecificationFactory specificationFactory,
                                                                    RDF
rdfFactory,

MappingDistinctTransformer
mappingDistinctTransformer) {
    this.mappingNormalizer =
mappingNormalizer;
    this.mappingSaturator = mappingSaturator;
    this.factConverter = inserter;
    this.mappingMerger = mappingMerger;
    this.settings = settings;
    this.sameAsInverseRewriter =
sameAsInverseRewriter;
    this.specificationFactory =
specificationFactory;
    this.rdfFactory = rdfFactory;
    this.mappingDistinctTransformer =
mappingDistinctTransformer;
}

```

And a constructor, let's say, for `RDFFactory` is

```
public class SimpleRDF implements RDF {  
  
    public SimpleRDF() {  
    }  
    ...  
}
```

As you see, in the end we reach a constructor without parameters.

In `IntelliJ`, the name of the properties file binding a specific interface can also be retrieved through the `find usages` functionality.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Intermediate Query \(IQ\)](#)

[Known issues →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▼

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

TDengine

Supported since 5.4.0.

Through the TDengine connector, Ontop is able to construct VKGs on TDengine databases.

Limitations & Exceptions

- As a time-series database, TDengine expects that all tables have a timestamp as their first column which is also used as a primary key.
- TDengine has limited support regarding date/time types. In particular, it does not support the `TIMESTAMP WITH TIME ZONE`, `DATE`, and `TIME` types and instead only uses `TIMESTAMP`
- Some data types, mainly `DECIMAL` and the `UNSIGNED` variants of integer types, are only supported through the WebSocket connection and are not currently supported by Ontop.

Database Connection

There are three ways to connect Ontop to a TDengine database using the JDBC driver:

- Native connection: depends on the client driver (libtaos.so on Linux, taos.dll on Windows, and libtaos.dylib on macOS). Below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the native driver:

```
jdbc.url = jdbc:TAOS://hostname:6030/dbname
jdbc.driver = com.taosdata.jdbc.TSDBDriver
```

sh

- WebSocket connection: does not require any additional client driver. Below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the WebSocket connection:

```
jdbc.url = jdbc:TAOS-WS://hostname:6041/dbname
jdbc.driver =
com.taosdata.jdbc.ws.WebSocketDriver
```

sh

- Restful connection: it is now recommended to use the WebSocket connection instead, as it achieves better performance. Nevertheless, it is still supported so below is an example of a `.properties` file that can be used to connect Ontop to TDengine using the Restful connection:

```
jdbc.url = jdbc:TAOS-RS://hostname:6041/dbname
jdbc.driver = com.taosdata.jdbc.rs.RestfulDriver
```

sh

NOTE

While all three connection methods are supported, the WebSocket connection is the recommended one, as it provides the best performance without any client compatibility issues.

Nested Type Support

TDengine has some limited support for JSON data through the `JSON` data type. The main limitation is that the JSON data type can only be used in tag columns of supertables, and it cannot have a nested structure (i.e., arrays or objects inside the JSON are not allowed).

Ontop does not support nested data types or `flatten lenses` for TDengine.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← SparkSQL](#)

[Trino →](#)

Guide

- Introduction
- Key concepts
- Getting started
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▾

- Generic JDBC (not recommended)
- AWS Athena
- AWS DynamoDB
- AWS Redshift
- Databricks
- DB2
- Denodo
- Dremio
- DuckDB
- Google BigQuery
- H2
- MariaDB
- MS SQL Server
- MySQL
- Oracle
- PostgreSQL

Snowflake

Supported since 5.0.0.

Through the [Snowflake](#) connector, Ontop is able to construct VKGs on cloud-based Snowflake databases.

Limitations & Exceptions

WARNING

Integrity constraints are often missing in Snowflake. Make sure to specify the missing ones in order to avoid very inefficient queries. We recommend using lenses for this purpose.

WARNING

If you want to use the Snowflake JDBC Driver with Java 17, the following option must be added to the JVM `--add-opens java.base/java.nio=ALL-UNNAMED`.

Alternatively, you can add the parameter `JDBC_QUERY_RESULT_FORMAT=JSON` to the JDBC URL to avoid using Apache Arrow.

- Due to an issue in the JDBC, quotation marks cannot always be used when defining an alias. Because of this, it is recommended to choose column names without special characters where possible.
- Accessing struct fields with the "colon operator" is not supported (see below).

Database Connection

The following shows the content of a sample `.properties` file that can be used to connect Ontop to Snowflake:

```
sh
jdbc.url =
jdbc:snowflake://dummyuser.snowflakecomputing.com/?
warehouse=dummywarehouse&db=dummydb&schema=dummyschema

jdbc.user = user
jdbc.password = password
jdbc.driver =
net.snowflake.client.jdbc.SnowflakeDriver
```

Nested Type Support

Ontop implements explicit compatibility with the `ARRAY` type, which can be used with the `flatten lens`. However, it is not able to automatically infer the type of the resulting column.

WARNING

Due to an issue mentioned above, using special characters inside the `output` and `position` fields of the flatten lens for Snowflake will fail. Furthermore, these fields will always be case-insensitive.

Struct Access

Individual object fields can be accessed by SQL expressions in the dialect's default way:

```
SELECT my_object['my_field'] FROM ...
```

WARNING

Accessing object fields using the "colon operator" is not supported. Please use the `[]` operator instead.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Presto](#)

[SparkSQL →](#)

Extending VKG Systems with RDF-star Support

by

Lukas Sundqvist

Supervisor: Prof. Diego Calvanese

Co-Supervisor: Dr. Benjamin Cogrel

A document submitted in partial fulfillment of the requirements for the degree of

MSc in Computational Data Science

at

FREE UNIVERSITY OF BOZEN-BOLZANO

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Professor Diego Calvanese, and Benjamin Cogrel. They have both served as invaluable advisors and tutors throughout the work of this thesis.

I also want to thank my classmates. Our “Fellowship of the data” kept studying fun and on track during the COVID-19 pandemic, and through the strict Italian lockdown.

And last but not least I would like to thank my loving partner and family for supporting me during this time.

ABSTRACT

This thesis concerns an extension to Virtual Knowledge Graph (VKG) systems, a state of the art technique of data integration. These systems perform data integration through the mapping of relational data sources to a virtual knowledge graph. The graph is often expressed using the W3C standard of Resource Description Framework (RDF), in this case the graph contains data in the form of triples, also known as facts.

RDF-star is a proposed extension to RDF that enables the annotation of RDF triples. It allows for a further layer of contextual data that until now has been difficult to implement using standard RDF. As RDF-star is a current field of research with few publications to its name this thesis contains a description of the proposal.

We investigate how to extend VKG systems with RDF-star support in order to enhance their support for contextual data. To this end we identify the mapping as the VKG component most crucial to extend. This is the component that specifies how to generate the virtual knowledge graph from the data contained in the data sources.

In order to extend the mapping we propose an extension to the W3C standard mapping language R2RML that we call R2RML-star. This allows for generating RDF-star data from relational data sources, enhancing the system's virtual knowledge graph with all the benefits of RDF-star in regard to contextual data handling.

We detail all the ways a VKG system must be extended in order to handle our proposed extension. This includes support for querying the generated RDF-star data, which is done through the use of SPARQL-star, an extension of the W3C standard query language for RDF data, SPARQL.

As a proof-of-concept we implement our proposed changes in the state of the art VKG system Ontop. We describe in detail the challenges this posed and how we overcame them, hoping to serve as inspiration for further development in this field.

Contents

1	Introduction	1
1.1	Virtual Knowledge Graphs	1
1.1.1	The problem of contextual data	2
1.1.2	The VKG system Ontop	3
1.1.3	Goals	3
1.2	Main Contributions	3
1.2.1	RDF-star	3
1.2.2	Extending VKG systems with RDF-star support	4
1.2.3	Extending Ontop with RDF-star support	5
1.3	Thesis Structure	5
2	Background	7
2.1	Resource Description Framework	7
2.2	Knowledge Graphs	8
2.3	SPARQL	8
2.3.1	SPARQL query language	8
2.3.2	Answer formats	10
2.3.3	SPARQL algebra	10
2.4	Relational Databases	11
2.5	Ontologies	12
2.5.1	RDFS	13

2.5.2	OWL2 QL	13
2.5.3	SPARQL entailment	15
2.6	Virtual Knowledge Graphs	16
2.6.1	Mapping	16
2.6.2	Formal description	18
2.6.3	VKG answering process	19
3	RDF-star	23
3.1	Motivation	23
3.2	Extending RDF's Abstract Syntax	24
3.3	Concrete RDF-star Syntaxes	26
3.3.1	Turtle-star	26
3.3.2	TriG-star, N-Triples-star, N-Quads-star	27
3.4	SPARQL-star	27
3.4.1	Algebra	28
3.4.2	Functions and operators	29
3.4.3	Answer formats	29
4	Investigating RDF-star Support in VKG Systems	31
4.1	RDF-star Support in the Mapping	31
4.1.1	R2RML extension	32
4.2	SPARQL-star Support	34
4.2.1	Parsing SPARQL-star queries	34
4.2.2	Extending query rewriting	34
4.2.3	Extending query unfolding	35
4.2.4	Extending query answers	36
4.2.5	Adding support for new functions and operators	37

4.3	End to End Example	37
4.3.1	The data integration scenario	37
4.3.2	A standard RDF solution	37
4.3.3	A R2RML-star mapping	39
5	Implementing RDF-star Support for Ontop	41
5.1	Parsing the Query	41
5.1.1	RDF4J's parsing of SPARQL-star queries	41
5.2	Intermediate Query	42
5.2.1	Query nodes	43
5.3	Rewriting SPARQL-star Queries in Ontop	45
5.4	Unfolding SPARQL-star Queries in Ontop	45
5.4.1	Motivation for matching	46
5.4.2	Translation from R2RML-star to IQ	47
5.4.3	Conversion from intensional to extensional data node	50
5.5	Supporting SPARQL-star Answers in Ontop	52
5.6	Supporting SPARQL-star Functions and Operators	52
6	Conclusion	53
6.1	Results	53
6.1.1	Theoretical investigation	53
6.1.2	Extending Ontop	53
6.2	Further Work	54
6.2.1	Further theoretical work	54
6.2.2	Further implementation work	54
	Reference List	55
	Reading List	59

Appendices	60
A RDF Graphs	60
B R2RML Mappings	60

1 Introduction

In this chapter we introduce the reader to the thesis. We present the thesis' background, goals, and main contributions to research. At the end of this chapter we present an overview of the thesis' structure.

1.1 Virtual Knowledge Graphs

This section introduces the background, problem statement, and goals of the thesis. The full background can be read in the first non-introductory part of our work, Chapter 2.

Our work builds on the popular *Virtual Knowledge Graph* (VKG) approach to data integration and it is therefore necessary to introduce this approach in order to describe the remainder of the thesis. Most common data integration techniques in use today follow an approach where the data must be materialised into a new database. This is the case for the data lake and data warehouse settings, which see widespread use in organizations today [19].

Materialising the data sources into a new integrated repository comes with several drawbacks. For one, if the data sources change then they must once again be loaded into the integrated repository, which creates a risk of hosting outdated data if updates are not run frequently enough. Second, the materialisation duplicates data, which requires additional hardware resources. Alternatively the original data sources can be removed after integration, but this means the originals are lost. This is often not an option e.g. because of system or legal constraints. It also requires that the integrator has the relevant access to the data sources and permission to modify them, which is not always the case [1, 19]

A virtual approach to data integration leaves the data sources as they are and does not create a new one. Instead, an application runs as a middle layer between the data sources and the end user, allowing user posed queries to be executed over all the data sources and return integrated answers [19]. The Virtual Knowledge Graph approach is, as the name implies, one such virtual solution.

Using the VKG approach, data sources are integrated into a virtual graph database, a Knowledge Graph, that is queried by the end user using a graph query language. A *mapping* describes how data in the data sources are mapped to elements in the knowledge graph. Optionally an *ontology* is used to give ontological information and enrich query answers.

While running, the VKG system takes a user posed query and, if possible, *rewrites* it with regard to the supplied ontology in order to capture more answers. The query

is then *unfolded* with regard to the given mappings and is then ready to be executed over the data sources in their respective query languages. After the answers have been retrieved, they are compiled and returned to the user. From the end user's perspective, they are just querying a graph. The actual functioning of the virtual approach is hidden from them.

1.1.1 The problem of contextual data

The VKG approach suffers from an issue regarding contextual data. The knowledge graph itself is often expressed using the W3C standard language of *Resource Description Framework*, or RDF for short¹. In this language one data point is expressed as a triple of *subject*, *predicate*, and *object*. We call such a triple a *fact*.

The problem arises when we want to comment our data, or when we want to add context by annotating the facts. In standard RDF this is already supported but through a very cumbersome technique known as RDF reification [27]. This technique does not work well for the VKG approach, as we show in Section 3.1. Other techniques have been suggested to deal with the metadata issue including the use of singleton properties and named graphs. As shown by Hartig, these techniques have their own shortcomings and do not on their own solve the contextual data issue, motivating the development of a new solution [27].

RDF-star is such a solution, developed by Hartig and a W3C Community Working Group under his lead [15, 27]. It extends the RDF standard and allows embedding a full RDF-triple in the subject or object position of another triple. We detail the RDF-star extension and work related to it in Chapter 3. Let us proceed with an example to clear things up.

Example 1.1 We wish to express the fact that Lukas Sundqvist is a student of the Free University of Bozen-Bolzano. This can be represented by the triple:

```
(:people/LukasSundqvist, :studentOf, :institutions/UniBZ)
```

Here the subject of the triple is an object representing Mr Sundqvist, coincidentally the author of this thesis. The predicate is the property `:studentOf`, and the object is an object representing the university.

Imagine now that we want to annotate this fact with some context, for example its source. As I just wrote this triple down in the thesis I'll use myself as source. Using a concrete RDF-star syntax, the annotated triple could be expressed as:

```
<< :people/LukasSundqvist :studentOf :institutions/UniBZ >>
   :source :people/LukasSundqvist .
```

Here the original triple is *embedded* or *quoted* inside a root triple where it plays the role of subject. ◇

Improving the support for contextual data in RDF can have a great impact on VKG systems using RDF to express their knowledge graph. In a data integration scenario,

¹<https://www.w3.org/RDF/>

contextual data plays an important role, for instance as in the example above one can annotate triples with a source property to indicate from which data source they were generated.

1.1.2 The VKG system Ontop

Ontop² is a actively developed state of the art VKG system [1, 2]. It is an open source project developed by researchers and Ontopic³, a company offshoot of the Free University of Bozen-Bolzano. Ontop currently lacks any support for RDF-star, like all current VKG systems, but plays a large role in this thesis as proof of concept of our work.

The system uses W3C standards for all the parts of the VKG approach. RDF for the knowledge graph, R2RML for the mapping, and OWL2 QL and RDFS as ontology languages. This means that extending these standards to allow RDF-star to play a role in VKG systems allows us to also practically implement and extend Ontop.

R2RML is the W3C standard language for mapping data from relational databases to graphs, and one of the mapping languages supported by Ontop. It allows for mapping relational data to an RDF graph. The mapping itself consists of a set of mapping assertions, and each such assertion is split up into two parts: a source part that is a SQL query, and a target part consisting of *triple templates* showing how to generate RDF facts from the source data delivered through the SQL query. To our knowledge there is no research yet into extending this language to support the generation of RDF-star data.

1.1.3 Goals

Following from our description of the current state of the art in this field, there are three main goals of this thesis. The first is to give a complete description of RDF-star, the second is to investigate how to extend VKG systems with RDF-star support from a theoretical point of view, and the third is to implement this support in the VKG system Ontop.

1.2 Main Contributions

In this section we outline our main contributions to research.

1.2.1 RDF-star

The first main chapter details the current state of RDF-star. In it we expand on the motivation behind RDF-star followed by a description of RDF-star and related extensions to other standards.

²<https://ontop-vkg.org/>

³<https://ontopic.ai/en/>

We first describe RDF-star’s abstract syntax as an extension of RDF’s abstract syntax, followed by extensions to the concrete RDF syntaxes such as Turtle and N-Triples. We then present a section on SPARQL-star, the accompanying extension to the standard graph query language for RDF graphs, SPARQL. We touch on details of SPARQL-star and how it differs from SPARQL such as its changed basic definitions, its algebra, new functions and operators, and new answer formats.

1.2.2 Extending VKG systems with RDF-star support

The second main chapter contains our investigation into adding RDF-star support to VKG systems. We deduce which is the most important VKG component to extend, namely the mapping, and do so by extending the W3C standard language for mappings, R2RML. The first step is to expand the concept of mapping in the VKG context such that it allows for the generation of RDF-star triples, we do so by introducing the concept of *triple-star template*.

In a similar manner to how RDF-star extends RDF, we propose to extend R2RML by means of extending the target part of the mapping. Our extension necessarily breaks the R2RML standard, and mappings written in the language of our extension will not function in an unsupported system. This is actually a desirable property as it brings attention to the user that something is wrong and does not result in silent errors.

By means of a new vocabulary, the R2RML concepts of TripleMap and TermType are extended. We suggest a new RDF-star TermType that allows the user to describe in the mapping a template for generating RDF-star triples. The new mapping language, which we call *R2RML-star*, is designed to ensure that only valid RDF-star triples are created from the new type of template.

The new mapping language has effects on the query processing of the VKG system. The main parts of processing are the query rewriting and unfolding processes, and we detail how they are affected by the introduction of R2RML-star and how VKG systems are extended to handle this new functionality.

We describe how to extend VKG systems to support the accompanying standard of SPARQL-star. As a VKG system is a query answering system, the query language should support the knowledge graph’s data model. If the graph is populated with RDF-star data, SPARQL-star support is required to correctly query it. Strictly speaking a VKG system can have various degrees of SPARQL-star support, having limited support limits the types of answers the end user can receive to their query. This is also detailed in the chapter.

In our view, a VKG system that supports SPARQL-star and R2RML-star mappings has been extended with RDF-star support. It allows the system to populate the knowledge graph with RDF-star contextual data and allows for querying the graph.

1.2.3 Extending Ontop with RDF-star support

Finally, much of the work done during this thesis has concerned a practical implementation of our suggested RDF-star extensions to the VKG system Ontop. This process is detailed in the third and last main chapter.

Based on our theoretical discussion on extending VKG systems, we detail how this was put in practice to implement the extensions in the open source Ontop project. Descriptions of Ontop's full query processing system are followed by a discussion on how to best change them to accommodate for the R2RML-star proposal. Internally in Ontop a query is represented with a tree like data structure called the Intermediate Query or IQ for short. The IQ is intended to bridge the gap between SPARQL and SQL queries and is the internal structure to which all query processing is applied. Much of our work has been focused on this data structure and how to extend its functionality so that rewriting and unfolding with regard to the R2RML-star proposal works as intended.

Through the discussion of this chapter we show that our implementation performs well and gives the expected results, this is verified through testing. The chapter is a valuable resource for learning about how to implement RDF-star support for other VKG systems, how to continue the work we have done on Ontop, or as a general developer's guide to Ontop for those interested in the systems internal workings.

1.3 Thesis Structure

Here follows an outline of the thesis to aid the reader to navigate its structure. The chapter you have just read is the introduction and it is followed by Chapter 2, the background chapter. In it we present to the reader the current state of research related to Virtual Knowledge Graphs. We present the relevant components of a VKG system, namely the ontology, mapping, knowledge graph, and graph query language. We also describe the relevant W3C standards including: RDF for the knowledge graph, R2RML for the mapping, RDFS and OWL2 QL for the ontology, and SPARQL for the query language.

The next three chapters make up the main chapters of the thesis. They begin with Chapter 3 on RDF-star. We explain the motivation behind RDF-star, its abstract and concrete syntaxes, and SPARQL-star in detail.

Chapter 4, investigating RDF-star support in VKG systems, is where we propose our RDF-star extension to VKG systems. We define the concept of RDF-star mapping and put it into practice by defining an extension to the mapping language R2RML that we call R2RML-star. We go through integrating SPARQL-star and how a VKG system must be adapted through all the stages of query processing to handle both SPARQL-star and R2RML-star mappings. We end on an example, showing the usage of our proposed extension.

In Chapter 5, dealing with the implementation of RDF-star support, we describe how we implemented this extension in the open source project Ontop. This includes

1 Introduction

details on the internal workings of Ontop and how functionality was added in order to handle SPARQL-star and R2RML-star mappings.

The last chapter is the conclusion, Chapter 6. Here our results are summarised and we discuss further work to be done in the field. Our references and appendices are then found at the end of the thesis. The appendices contain the full versions of RDF graphs and R2RML mappings used as examples throughout the thesis.

2 Background

In this section prerequisite knowledge is presented to the reader. Readers that are familiar with one or more of these subjects can freely skip sections.

The main prerequisite knowledge required for this thesis is an understanding of the Virtual Knowledge Graph (VKG) approach to data integration. To this end, we first explain all the components of a VKG system: the knowledge graph, query language, ontology and mapping; before finally explaining the system itself. We assume that the reader is familiar with basic notions of Computer Science, e.g. as taught in a Bachelor's degree in Computer science.

2.1 Resource Description Framework

The Resource Description Framework (RDF) is a framework to describe information on the web, standardized through a collection of W3C recommendations [4]. It is the data model underlying the Semantic Web and the VKG approach to data integration [1].

The basic building blocks of this data model are known as *RDF terms*. An RDF term is either an *IRI*, *blank node*, or *literal*. An IRI is a Unicode string with certain restrictions used to uniquely identify a resource on the web. An IRI looks very much like a URL. Blank nodes are local identifiers used in concrete RDF syntaxes if an identifier is unknown. Literals are values such as strings or integers that do not need to be uniquely identified [4].

An *RDF triple* consists of three components (*subject*, *predicate*, *object*). Seeing the subject and object as two nodes, the predicate forms a directed labeled edge from subject to object. Hence, a set of triples represents a directed edge-labeled graph [4].

Definition 2.1 Assume that $\mathcal{I}$ is the set of IRIs, $\mathcal{B}$ the set of blank nodes, and $\mathcal{L}$ the set of literals. An *RDF triple* is a tuple (s, p, o) , consisting of a subject $s \in \mathcal{I} \cup \mathcal{B}$, a predicate $p \in \mathcal{I}$, and an object $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ [4]. ♦

Definition 2.2 A set of RDF triples is an *RDF graph* [4]. ♦

When writing RDF graphs, a concrete syntax is needed. For this background chapter, the Turtle syntax is always used unless another syntax is specified [14]. For further information on RDF it is recommended to read the RDF specification [4].

Example 2.3 Consider the IRIs:

- $i_1 = \text{http://thesis.org/data/Lukas\%20Sundqvist}$
- $i_2 = \text{http://thesis.org/knows}$
- $i_3 = \text{http://thesis.org/age}$

Along with the blank node b_1 and the literal $l = 28$. Then (i_1, i_2, b_1) and (b_1, i_3, l) both form RDF triples. Together they form an RDF graph describing that the author of this thesis knows someone who's age is 28. $\diamond$

2.2 Knowledge Graphs

Knowledge graphs have been vaguely defined by academics, practitioners, and companies for several years, which makes it difficult to pin down an exact definition. Färber et al. define knowledge graphs (KGs) simply as any graph written in the language of the Resource Description Framework (RDF) [18, p.1]. Ehrlinger on the other hand defines a KG as a system that “acquires and integrates information into an ontology and applies a reasoner to derive new knowledge.” [17]. This second definition could nearly describe the entire VKG approach to data integration.

The difficulty of formally defining a knowledge graph is testament to the complexity of the term, nevertheless they deserve a proper definition. We will use as base the definition by Färber et al. and are inspired by the work of Ehrlinger [17, 18] in defining KGs within the scope of this thesis.

Definition 2.4 A *Knowledge Graph* is an RDF Graph describing objects of a particular domain and how they relate to each other. $\blacklozenge$

Even though KGs are never materialized in the VKG approach, they play a major role as this is the shape of the final data presented to the end user for querying. This leads us to the query language that the end user of a VKG system uses to retrieve data.

2.3 SPARQL

SPARQL is a standard used for querying RDF data, similar to the more familiar SQL used to query relational data. It is formalized in a W3C recommendation that forms the basis of this section, and the interested reader can look there for more information [5]. SPARQL is used to query an *RDF dataset*, which consists of one or more RDF graphs. One of these is the unnamed default graph and the others are named by IRIs [6].

2.3.1 SPARQL query language

The SPARQL query language is used to write queries posed to RDF datasets. Each query starts with two important parts, at the top are prefix declarations allowing

terser syntax, and the second part sets the *query form*. SPARQL has four query forms which are:

- SELECT : Returns all or some of the variables bound.
- CONSTRUCT : Returns a new RDF graph based on a specified pattern.
- ASK : Returns a boolean indicating whether the query found a match or not.
- DESCRIBE : Returns an RDF graph describing the resources found [6].

Of these, the original, most used, and most important form is the SELECT query and that is the one we will be focusing on in this thesis. This form is roughly analogous to a SQL SELECT query and returns results in one of a number of answer formats, which will be further discussed in Section 2.3.2. A SELECT query returns all or a subset of variables bound during the pattern matching stage of executing the query.

The third part of the query is the WHERE clause, from where pattern matching is performed. This clause is analogous to a SQL WHERE clause with the main difference being that we are querying RDF data in the form of a graph in place of a relational database. Everything contained within the WHERE clause forms a *group graph pattern* that the dataset being queried has to fit in order to return results. The simplest group graph patterns are called *basic graph patterns* and they are made up only of *triple patterns*. Triple patterns consist of RDF terms and *SPARQL query variables* which get bound if part of the queried graph matches the pattern [6]. This is illustrated in Example 2.7. We will formally define most of these terms now, for further reading, see the SPARQL standard [5].

Definition 2.5 Let $\mathcal{I}$ be the set of IRIs, $\mathcal{V}$ the set of SPARQL query variables, and $\mathcal{L}$ the set of RDF literals. Then a *SPARQL triple pattern* is a tuple (s, p, o) with

$$s \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}, \quad p \in \mathcal{V} \cup \mathcal{I}, \quad o \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}.$$

Note that unlike the RDF standard, literals are allowed in the subject position [5].♦

Definition 2.6 A set of SPARQL triple patterns is a SPARQL Basic Graph Pattern (BGP) [5]. ♦

Group graph patterns can include more complex keywords such as OPTIONAL, JOIN, or FILTER, which complicates the semantics of the query. We refer the reader to the SPARQL standard for further information [5]. Following the WHERE clause is the fourth part of the query, containing familiar keywords analogous to their SQL counterparts, such as GROUP BY, ORDER BY, or LIMIT [6].

Example 2.7 Recall the RDF graph described in Example 2.3. Assume that it is the only and default graph forming our RDF dataset. Consider the SPARQL query:

2 Background

```
PREFIX thesis: <http://thesis.org/>
SELECT ?person
WHERE {
  ?person thesis:knows _:b1.
  _:b1 thesis:age 27.}
```

Let us work through the query step by step. The PREFIX declaration allows us to save work writing the rest of the query. Wherever **thesis:** later appears it is replaced with the thesis IRI by the program executing the query. The SELECT clause signifies the query form and is followed by the variables that will be returned to the user. The WHERE clause specifies the group graph pattern to be matched, which in our case consists of two triples.

The triple patterns of the query match to the two triples of our RDF dataset binding the variable **?person**. There is only one match and therefore only one result is returned: **?person** is bound to **http://thesis.org/data/Lukas%20Sundqvist**.◇

2.3.2 Answer formats

The result of a SPARQL SELECT query is returned through one of several standardized formats representing mappings between the variables of the SELECT clause and the RDF-terms they have matched to [5]. These formats are:

- XML, the original format which is standardized in [7].
- JSON, standardized in [8]
- CSV and TSV, standardized in [10].

2.3.3 SPARQL algebra

The grammar and syntax of a SPARQL query are designed for ease of use and for humans writing queries. A translation into a more formal SPARQL algebra is possible and defined by the SPARQL standard [5]. The definitions and algebra in this section are mainly sourced from the seminal work by Pérez, Arenas, and Gutierrez [26]. We describe only a simplified SPARQL algebra, which is needed to understand the work done in this thesis. For additional reading please see the SPARQL standard and the paper by Pérez et al [5, 26]. To start, we define the notion of graph pattern expression.

Definition 2.8 We define a *SPARQL graph pattern expression* recursively as follows [26].

- A SPARQL triple pattern is a graph pattern expression.
- If P_1 and P_2 are graph pattern expressions, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPTIONAL } P_2)$, and $(P_1 \text{ UNION } P_2)$, are all graph pattern expressions.

- If P is a graph pattern expression and R is a SPARQL *built-in condition*, then $(P \text{ FILTER } R)$ is a graph pattern expression.

A SPARQL built-in condition is a boolean function such as equality, bound, or isIRI. For further information we refer the reader to Pérez et al. and the SPARQL standard [5, 26]. $\blacklozenge$

Definition 2.9 A *SPARQL solution mapping* is a partial function μ mapping query variables to RDF terms, $\mu : \mathcal{V} \rightarrow \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, where $\mathcal{B}$ is the set of blank nodes. The *domain* of μ is called $\text{dom}(\mu)$. We denote by $u(t)$ the triple obtained through substituting the variables in the triple pattern t according to the solution mapping μ . We say that μ_1 and μ_2 are *compatible* if $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ implies that $\mu_1(x) = \mu_2(x)$ [26]. $\blacklozenge$

Definition 2.10 Let Ω_1 and Ω_2 be sets of SPARQL solution mappings. The *join*, *union*, *difference*, and *left join* are defined as follows [26].

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\}.$
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \cup \Omega_2\}.$
- $\Omega_1 \setminus \Omega_2 = \{\mu \mid \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are incompatible}\}.$
- $\Omega_1 \Join \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2).$ $\blacklozenge$

Definition 2.11 Let D be an RDF dataset, tp a triple pattern, and P_1 and P_2 graph pattern expressions. The *evaluation* of a graph pattern over D , call it $\llbracket \cdot \rrbracket_D$, is a function from the set of graph pattern expressions that returns a set of solution mappings. It is defined as follows [26].

- $\llbracket tp \rrbracket_D = \{\mu \mid \text{dom}(\mu) = \text{var}(tp), \mu(tp) \in D\}$, with $\text{var}(tp)$ being the set of variables occurring in tp .
- $\llbracket T_1 \text{ AND } T_2 \rrbracket_D = \llbracket T_1 \rrbracket_D \bowtie \llbracket T_2 \rrbracket_D.$
- $\llbracket T_1 \text{ OPTIONAL } T_2 \rrbracket_D = \llbracket T_1 \rrbracket_D \Join \llbracket T_2 \rrbracket_D.$
- $\llbracket T_1 \text{ UNION } T_2 \rrbracket_D = \llbracket T_1 \rrbracket_D \cup \llbracket T_2 \rrbracket_D.$ $\blacklozenge$

2.4 Relational Databases

Relational databases play an important role in this thesis but knowledge of them is assumed to be background knowledge to the reader. If this is not the case we recommend a classic book on database theory such as Abiteboul, Hull, and Vianu [21]. Nonetheless relational data normally makes up the data sources being integrated in the VKG approach and we will now set up an example that will be frequently used throughout the thesis.

2 Background

IMDB database	imdbTable1	
Name	Year	Score
The Shawshank Redemption	1994	9.2
The Godfather	1972	9.2
Pulp Fiction	1994	8.9

Table 2.1: Description of the first database, containing data sourced from the Internet Movie Database.

Rotten Tomatoes database	RtTable 1	
MovieName	ReleaseYear	Rating
A Star is Born	2018	0.78
A Star is Born	1937	0.79
The Godfather	1972	0.98

Table 2.2: Description of the second database, containing data sourced from Rotten Tomatoes.

Example 2.12 Consider a movie theater holding information about movies in two different relational databases, each containing one table. The databases and the data stored therein are shown in Tables 2.1 and 2.2. We note that the primary key is composed of the Name and Year (or MovieName and ReleaseYear) columns, as the name alone is not a unique identifier.

The problem of data integration in this specific case concerns integrating the two databases so that we can query them both at the same time. $\diamond$

2.5 Ontologies

The notion of *ontology* has its roots in philosophy, but we consider here the term as it is used in the field of Computer Science. In this field it represents the relevant notions in a domain or universe of interest, for example the domain of films which is our running example in this thesis.

Kaufmann (2012) defines an ontology as “a description of the entities in [a specific] domain, relationships between them, and any other known constraints.” [19, p.325] and we will narrow this down further to define an ontology for the purpose of this thesis.

Specifically we consider ontologies that have their roots in the world of description logics. Such an ontology consists of an *A-Box* a *T-Box*, with the A and T representing assertions and terminology respectively. In the A-Box fit all those statements that describe individuals, while the T-Box contains all statements with general information about the domain. Using this language one splits an ontology in two sets of statements, assertions and terminology. For an example, imagine once again the Hollywood domain. The statement that Tom Hanks is an actor is an assertion and

belongs in the A-Box, while the statement that directors direct movies is part of the T-Box [20].

When using the VKG approach for data integration, the ontology tends to only or mainly contain terminology, i.e. statements belonging to the T-Box. For the language describing the ontology, Ontop supports OWL2 QL or RDFS, both written in the form of an RDF graph [1, 22].

Definition 2.13 An *ontology* is an RDF graph describing the terminology of a certain domain, meaning abstract entities, relationships between them, and any other known constraints. The ontology is written using one or both of the OWL2 QL and RDFS ontology languages. ♦

2.5.1 RDFS

RDF Schema or RDFS is a semantic extension of RDF, which can also be seen as a lightweight ontology language. For more detail see the W3C specification [11]. RDFS allows for:

- Typing, to say that something is a resource or a class. Or that something is a member of a specific class.
- Subclasses, where A is subclass of B means that all objects of class A are also objects of class B.
- Specifying range and domain of properties. E.g. one can say that only humans have an age, and the age must be an integer.

Some example statements using rdfs, assuming prefixes already defined:

```
:Person rdf:type rdf:Class
:Man rdfs:subClassOf :Person.
:stars rdf:type rdf:Property.
:stars rdfs:domain :Films.
:stars rdfs:range :Actor
```

2.5.2 OWL2 QL

OWL2 is a much more expressive ontology language that allows many different types of statements to be made. For this thesis we only make use of OWL2 QL which is a subset of OWL2 developed specifically for database querying [13]. OWL2 QL allows for more detailed information than RDFS, such as some integrity constraints on properties. For example: one can specify that each Film must have a Director.

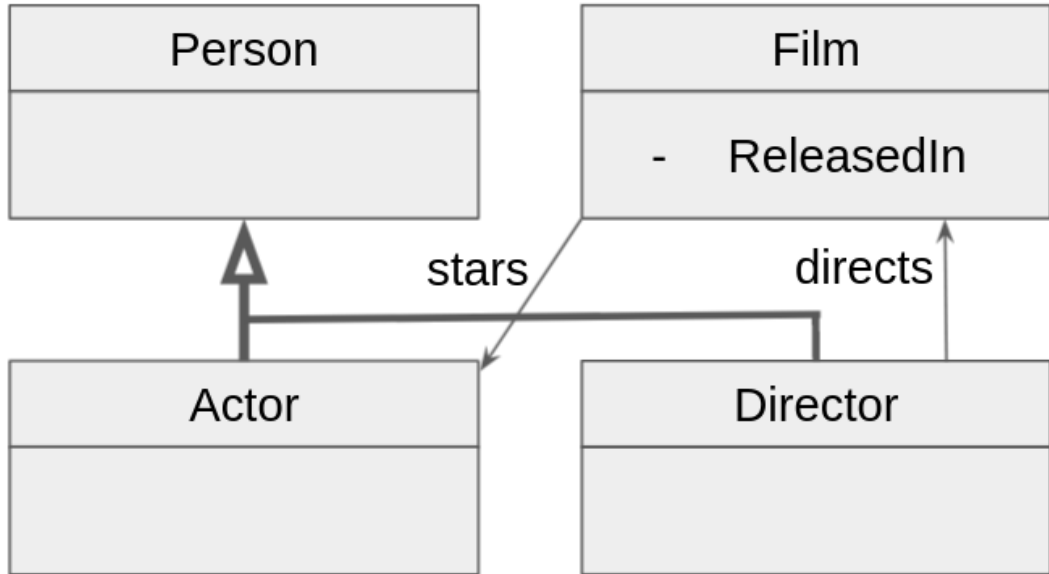


Figure 2.1: The ontology of Example 2.14 visualized as a UML diagram. Boxes represent classes, big arrows show subclass relations, small arrows are other relations between classes, and the 'releasedIn' signifies a relation to a literal.

Example 2.14 Staying with our film example, imagine that we want to describe the domain surrounding film. To start, we could describe the following notions.

- There are different types of objects: Persons, films, actors, directors.
- There are relations between these objects: A film stars actors and is directed by directors. Clearly actors and directors are people.
- There are relations between the objects and literals, for example Films have a year of release. ◇

This bare-bone ontology is described visually in Figure 2.1. Here the ontology is expressed in the Turtle syntax using the RDFS and OWL2 QL ontology languages.

```
<http://lukas.thesis.org/films> rdf:type owl:Ontology .
```

```
# Object Properties
:directedBy rdf:type owl:ObjectProperty ;
    rdfs:domain :Film ;
    rdfs:range :Director .
:directs rdf:type owl:ObjectProperty ;
    rdfs:inverseOf :directedBy ;
    rdfs:domain :Film ;
    rdfs:range :Director .
:stars rdf:type owl:ObjectProperty ;
```

```

    rdfs:domain :Film ;
    rdfs:range :Actor .

# Data properties
:releasedIn rdf:type owl:DatatypeProperty ;
    rdfs:domain :Film .

# Classes
:Actor rdf:type owl:Class ;
    rdfs:subClassOf :Person .
:Director rdf:type owl:Class ;
    rdfs:subClassOf :Person .
:Film rdf:type owl:Class .
:Person rdf:type owl:Class .

```

2.5.3 SPARQL entailment

SPARQL entailment is a way to reason over SPARQL datasets based on a set of rules. Entailment allows a SPARQL query to return answers that are not directly present in the queried graph, but that are inferred based on the information contained in the ontology. We start with an example courtesy of the W3C [12].

Example 2.15 Consider the following RDF graph:

```

ex:book1 rdf:type ex:Publication .
ex:book2 rdf:type ex:Article .
ex:Article rdfs:subClassOf ex:Publication .
ex:publishes rdfs:range ex:Publication .
ex:MITPress ex:publishes ex:book3 .

```

We wish to execute the following SPARQL query over the dataset consisting of the single RDF graph above:

```

SELECT ?prop
WHERE { ?prop rdf:type rdf:Property }

```

Under SPARQL rules as we know them this query will not match any data in the dataset and will return an empty answer, this is called *simple entailment*. However if the SPARQL query is executed under *RDF entailment* the system correctly deduces that `ex:publishes` is of type `rdf:Property`, seeing as it takes the place of predicate in the last line of the RDF graph. Next, consider the query:

```

SELECT ?pub WHERE { ?pub rdf:type ex:Publication }

```

Under simple or RDF entailment this query has a single answer: `ex:book1`. If we run the query under *RDFS entailment* it will return two additional answers. Due

to `ex:Article` being a subclass of `ex:Publication`, this implies that any articles, such as `ex:book2`, are also publications.

Also the third book, `ex:book3`, is returned as an answer under RDFS entailment. This is because it plays the role of object in a triple with `ex:publishes` as predicate, and because there is a RDFS range property specifying publishes to have publications as range. $\diamond$

The example above shows how a query answering system can use basic reasoning to populate a knowledge graph with more data than is explicitly stated. This is the process of SPARQL entailment and it is one of the key contributions of the ontology in the VKG setting.

Entailment regimes

In this section we have mentioned many *entailment regimes*. These are various rules for interpreting ontological information and extending the queried dataset to allow SPARQL to match answers to variables as intended. Examples of such regimes are RDF, RDFS, or OWL entailment regimes. For further reading we recommend to the reader the W3C Recommendation on SPARQL entailment [12].

2.6 Virtual Knowledge Graphs

The core of this thesis concerns an extension to the Virtual Knowledge Graph system Ontop, thus it is necessary to have a good knowledge of what such a system is, in order to understand the coming chapters. We are concerned with the use of *Virtual Knowledge Graphs* (VKG) in the field of data integration, something that is also called *Ontology-Based Data Access* (OBDA) in the literature [3].

In the case of using a VKG for data integration the data sources are usually relational databases. The data is then *virtualised* with the help of a *mapping* and an *ontology*, in order to create a virtual *knowledge graph* that is then *queried* by the end user [3]. Each of these components making up a VKG system have been described in previous sections, except for the mapping.

2.6.1 Mapping

In the context of data integration a *mapping* is a set of expressions describing a relationship between a set of schemata. Typically one considers mapping expressions from the schema of a data source to a mediated schema which is used to present integrated data from different data sources to the end user, using a common vocabulary [19].

In the VKG approach the ontology can be said to play the role of mediated schema. It establishes the terminology of the final knowledge graph to which the end user sends their queries. This means the mapping that we are interested in establishes a relationship between the schemata of the data sources on the one hand and the

ontology on the other hand [3]. It must therefore form a link between the relational and graph data models and be a way to construct graph data from relational data. As our focus in this work lies on the W3C standard graph data model RDF, we now define a RDF specific mapping concept.

Definition 2.16 Let Y be a set of columns that is the output of a SQL query, possibly empty. Consider three subsets, Y_1 , Y_2 , and Y_3 , of Y . Then $\mathcal{T}(Y)$ is called a *triple template*, and is composed of three functions $subject(Y_1)$, $predicate(Y_2)$, $object(Y_3)$. The output of these functions must match the respective expected type of a RDF triple. That is, recalling the definitions of $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ from Definition 2.1, it holds that

$$subject(Y_1) \in \mathcal{I} \cup \mathcal{B}, \quad predicate(Y_2) \in \mathcal{I}, \quad object(Y_3) \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}.$$

Defined in this way the triple template expresses how to use facts retrieved from a data source to construct RDF terms and triples. ♦

Again due to our focus on RDF graphs, we define mapping as it is used in our specific context.

Definition 2.17 A *mapping* in the VKG setting is a set of *mapping assertions*, each having the form $[\mathcal{T}(Y) \leftarrow (Y = sql(x))]$, where x is a data source, $sql(\cdot)$ is a SQL query over a data source schema, Y the output of that query, and $\mathcal{T}(Y)$ is a triple template. ♦

A VKG system uses the mapping to populate a virtual knowledge graph with facts sourced from the data sources. Merging the facts with the supplied ontology gives the complete knowledge graph that the end user queries. In order to describe mappings, Ontop supports both a native mapping language and the W3C recommendation R2RML. For the sake of readability we will use a simplified version of Ontop's native mapping language similar to that of Calvanese et al. in our examples [1]. At the same time we supply R2RML examples in the appendix.

Example 2.18 A mapping must relate one or more databases with the ontology, showing how to use the rows of a relational database to create RDF triples in the knowledge graph. Recall the IMDB database and the ontology of film described in Examples 2.12 and 2.14 respectively. Using a simplified mapping syntax, where $\{-\}$ denotes answer values from the SQL query, the mapping

```
:films/{Name}{Year} rdf:type :Film.
<- SELECT Name, Year FROM imdbTable1
```

applied to our example database gives rise to the triples

```
<http://f.org/films/The%20Shawshank%20Redemption1994> rdf:type :Film.
<http://f.org/films/The%20Godfather1972> rdf:type :Film.
<http://f.org/films/Pulp%20Fiction1994> rdf:type :Film.
```

2 Background

And, as part of a VKG system, this combination of components, mapping, ontology, and data source, ensures that these three triples are generated and take place in the knowledge graph. $\diamond$

R2RML

In order to express the mapping we need a mapping language, R2RML is the de-facto standard mapping language and a W3C recommendation [9]. R2RML is itself expressed in the form of an RDF graph and written down using the concrete Turtle syntax. It uses a custom vocabulary to define mapping assertions and their target and source parts [9].

Example 2.19 The mapping of Example 2.18 is expressed in R2RML by the following RDF graph:

```
<#TriplesMap1>
rr:logicalTable [ rr:tableName "imdbTable1" ];
rr:subjectMap [
  rr:template "http://lukas.thesis.org/films/{Name}{Year}";
  rr:class :Film;
];
rr:predicateObjectMap [
  rr:predicate ex:name;
  rr:objectMap [ rr:column "Name" ];
];
rr:predicateObjectMap [
  rr:predicate :releasedIn;
  rr:objectMap [ rr:column "Year"];
].
```

Where the `rr:` prefix signifies the R2RML vocabulary. For detailed explanations of the vocabulary we refer the reader to the R2RML standard [9]. $\diamond$

2.6.2 Formal description

Virtual Knowledge Graphs are formed by a combination of the components described so far. Formally VKGs have been specified well by Xiao et al. and we will use the same definitions for the purpose of this thesis.

Definition 2.20 A *VKG specification* is a tuple $P = (O, S, M)$, where O is an ontology, S is a data source schema, and M a mapping from S to O [2, 29]. $\blacklozenge$

Definition 2.21 A VKG specification $P = (S, P, O)$ together with a database D compliant with S forms a *VKG instance* (P, D) . Applying the mapping M to the data sources produces the RDF graph $M(D)$, and further applying reasoning over the ontology O on $M(D)$ produces the virtual knowledge graph associated with the instance: $G_{P,D}$ [2, 29]. $\blacklozenge$

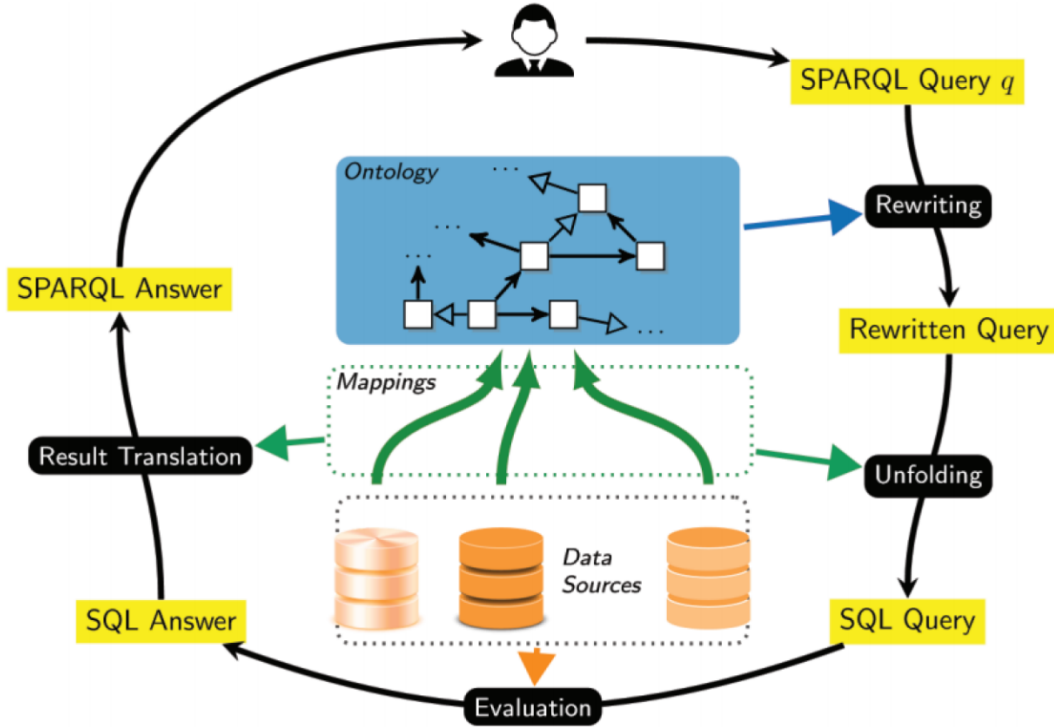


Figure 2.2: An overview of the VKG approach. Credit to Xiao et al. [2].

This graph $G_{P,D}$ is what is exposed to the end user who queries it using the standard language for querying RDF datasets, SPARQL. Taken together the two definitions above contain every necessary component and are sufficient for describing a VKG system.

2.6.3 VKG answering process

There are two large issues with how we have described the VKG approach until now. The first issue is that we have described individual components, but not much how they work together to integrate data. The second issue is that we have described a “bottom up” approach to data integration. We have described how mappings applied to data sources create RDF triples, which when combined with an ontology gives rise to a complete knowledge graph. This might be how the system appears to the end user, but the reality is that the VKG approach internally works completely different [1, 3]. This misunderstanding must be rectified and we take this section to reorient our understanding of the VKG approach.

One can divide the VKG approach to data integration into three parts, a manual configuration, an offline part, and an online part. The manual part is performed by a technical expert, it is the creation of the ontology and mappings. A very time consuming and difficult process requiring deep knowledge of every part of the system and domain, but a process that is out of the scope of this thesis and we will not

discuss it much further. The offline part follows when the system loads the created ontology and starts up. The online part is what the system does when it is running [1]. What happens from the point of receiving a query from the end user, to the point of returning results? Figure 2.2 illustrates this process.

The system starts its online processing as the user poses a SPARQL query. As the first step the query is rewritten with respect to the specified ontology. For example, if the query asks to retrieve all objects of type `:Person` and our ontology contains a triple specifying that `:Actor` is a subclass of `:Person`, the system adds to the query asking also for all actors.

The query is then unfolded using the mappings in order to move from SPARQL to SQL. Any reference to RDF terms in the query is checked against the mapping expressions to find where those terms are mentioned in the target part. The RDF terms in the query are then replaced by the corresponding source SQL query, generating one or more SQL queries over the data sources. The generated SQL queries are executed over the relational data sources, and the result is translated back to a SPARQL answer using the mapping in the conventional way [2].

In the process of rewriting and unfolding the query the VKG system Ontop uses an intermediate data structure called the Intermediate Query (IQ) to bridge the gap between the SPARQL algebra and relational algebra [3]. Much of the work of this thesis focuses on this area of Ontop's internal workings, and we will describe it in a later chapter.

Example 2.22 Let us combine all the previous parts of this chapter in order to give an example of a working VKG system. Let the system be set up with the data sources, ontology, and mapping as described in the previous sections.

Let the end user pose a simple SPARQL query asking for all objects that are films. The first step in the process is the rewriting of the query. In our case with a very bare bones ontology nothing will happen. If one wants to imagine how this step looks, think of an ontology stating that all films are movies and vice-versa. Our query would then be rewritten to ask both for films and movies. We are now in the Intermediate Query (IQ) stage.

The next step is the unfolding with regards to the mapping. In our case, we will use Mapping 1 of Example 2.18 and see that in order to answer queries asking for films, we must turn our attention to the database described in Table 2.1. The IQ will add a node describing a SQL query over this database.

Finally, all the SQL queries (in our case one) part of the IQ will be executed by the database systems of the data sources. This happens completely outside the VKG system. Once the relational databases return their answers Ontop takes over again, using the mappings to transform the answers into RDF triples. The final answers are then returned to the user that posed the original query.

Assuming the query was a simple SPARQL SELECT query asking for all films, since we only had the one mapping, the answers would be:

- `:TheShawshankRedemption1994`

- :TheGodfather1972
- :PulpFiction1994.



3 RDF-star

RDF-star is an extension of the RDF language proposed by a W3C community group¹. Its main contribution is to improve support for statement level metadata in RDF graphs through the use of quoted triples, also known as embedded triples [27].

In this chapter a motivation is presented for the extension, followed by a description of the abstract and concrete syntaxes of RDF-star and SPARQL-star.

3.1 Motivation

Adding metadata to RDF triples is already possible through the process of *RDF reification*, and in order to understand why RDF-star is needed we must first investigate how RDF reification works and where it falls short.

RDF reification is done using the specific RDF vocabulary of `rdf:Statement`, `rdf:subject`, `rdf:predicate`, and `rdf:object`. In combination with a blank node the vocabulary gives rise to the semantic meaning of a blank node representing a given RDF triple. In this way the blank node can then be used as subject or object in any other RDF triple [4].

Example 3.1 Consider an RDF triple representing that a film has a certain score. We would like to annotate this triple stating that the source of the statement is the Internet Movie Database. A way to do that through RDF reification is shown below.

```
:/data/TheGodFather1972 :hasScore "9.2".
_:a a rdf:Statement;
  rdf:subject :/data/TheGodFather1972;
  rdf:predicate :hasScore;
  rdf:object "9,2";
  :source "IMDB".
```

And in order to retrieve such data, consider this SPARQL query fragment:

```
SELECT ?film ?score ?source
WHERE {
  ?film :hasScore ?score.
  ?b a rdf:Statement;
    rdf:subject ?film;
    rdf:predicate :hasScore;
    rdf:object ?score;
    :source ?source.}
```

¹The latest draft can be seen at <https://w3c.github.io/rdf-star/cg-spec>

In addition to being very complex, the user must have knowledge specifically of RDF reification in order to formulate such a query correctly. $\diamond$

In short, for every triple one wishes to add metadata to through RDF reification, a blank node and three additional triples need to be introduced. As a result, both the RDF data and SPARQL query in the example above are written in a very convoluted way. If one needs to form more complex expressions, such as nesting meta comments, it further complicates things.

Other ways of adding statement level metadata to RDF graphs have been proposed in order to improve this situation. Named graphs and RDF singleton properties are two of these approaches, but they both suffer from their own flaws making them unsuitable for their purpose [27]. RDF-star aims to improve upon the state of the art by filling the function of adding metadata, also known as contextual data, to RDF triples in a concise manner [27].

3.2 Extending RDF's Abstract Syntax

The basis of RDF-star's contribution lies in the following definition.

Definition 3.2 Let $\mathcal{I}$ be the set of IRIs, $\mathcal{B}$ the set of blank nodes, and $\mathcal{L}$ the set of literals, and let $s \in \mathcal{I} \cup \mathcal{B}$ be a subject, $p \in \mathcal{I}$ a predicate, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ an object. Recall the RDF triple of Definition 2.1.

We define that any RDF triple also is an *RDF-star triple*. Further, if t and t' are RDF-star triples then (t, p, o) , (s, p, t) , and (t, p, t') , are RDF-star triples and t, t' are called *quoted triples* [15]. $\blacklozenge$

This definition is then naturally used to define several other terms. RDF terms, see Section 2.1, together with RDF-star triples form the *RDF-star terms*, and a set of RDF-star terms is called an *RDF-star graph* [15].

In this way the RDF triple that one desires to annotate simply takes the place of subject or object of an RDF-star triple, and the problems of other reification techniques are avoided. For example to add meta-data to a triple, only one additional triple needs to be introduced, in contrast with RDF reification where four additional triples must be added.

An RDF-star triple is uniquely identified by its three components of subject, predicate, object. It makes no difference where it is placed, as root triple or embedded. For example, assume $t = (s, p, o)$ is a triple and i an IRI; the triple (t, i, t) contains only one RDF-star triple but in two places [15].

A difference is made between quoted triples on the one hand and triples taking a place in an RDF-star graph. RDF-star triples that are part of an RDF-star graph are *asserted triples*, intuitively such a triple is a “root” triple. As an RDF-star triple is uniquely identified by its components, the same triple may be both asserted and quoted.

RDF-star is also backwards compatible with RDF. Through a *star-mapping*, unrelated to the mappings discussed previously in this thesis, RDF-star data can be converted to RDF data using the RDF reification vocabulary, and vice-versa [27].

Example 3.3 We wish to state the following fact expressed in English: The Internet Movie Database claims that the film The Godfather has a score of 9.2. Using the abstract RDF-star syntax, if we do not need to be very exact, this can be encapsulated by the triple: $(t, p_1, \text{"IMDB"})$, where

$$\begin{aligned} t &= (s, p_2, \text{"9.2"}), \\ p_1 &= \text{http://lukas.thesis.org/films\#source}, \\ s &= \text{http://lukas.thesis.org/films\#data/TheGodfather1972}, \\ p_2 &= \text{http://lukas.thesis.org/films\#hasScore}. \end{aligned}$$

In this example the root RDF-star triple contains another quoted RDF-star triple in the subject position. $\diamond$

Following Hartig [27] we further define some useful functions that, unlike what we have previously discussed, have no equivalent in the RDF standard.

Definition 3.4 The *elements* of an RDF-star triple t , denoted $\text{Elmts}(t)$, are all RDF-star terms appearing in the triple, or any quoted triples [27]. Mathematically speaking, if $t = (s, p, o)$ and $\mathcal{T}$ is the set of all RDF-star triples,

$$\text{Elmts}(t) = \{s, p, o\} \cup \{x \in \text{Elmts}(t') \mid t' \in \{s, o\} \cap \mathcal{T}\}.$$

The function is extended to an RDF-star graph G to be the union of $\text{Elmts}(t)$ [27], for all triples t in G :

$$\text{Elmts}(G) = \bigcup_{t \in G} \text{Elmts}(t).$$

Further we define the *triples of* an RDF-star graph G to be any RDF-star triples, embedded or not, contained in G [27]:

$$T(G) = G \cup (\text{Elmts}(G) \cap \mathcal{T}).$$

Note that following from the basic RDF-star definitions the sets defined here are all finite. $\blacklozenge$

Example 3.5 The elements of the triple t from Example 3.3 are

$\text{Elmts}(t) = \{p_1, p_2, s, \text{"IMDB"}, \text{"9.2"}\}$, where
 $p_1 = \text{http://lukas.thesis.org/films\#source}$,
 $p_2 = \text{http://lukas.thesis.org/films\#hasScore}$,
 $s = \text{http://lukas.thesis.org/films\#data/TheGodfather1972}$.

Note that t itself is not part of $\text{Elmts}(t)$. ◇

3.3 Concrete RDF-star Syntaxes

The RDF-star community group has also described extensions to several of the concrete syntaxes used in RDF documents.

3.3.1 Turtle-star

Turtle has been extended to allow quoted triples as subjects or objects forming the syntax of Turtle-star. The main way to represent a quoted triple is through the use of double less/greater than signs ($\langle\langle \text{ s p o } \rangle\rangle$), we refer to this as the *less/greater than syntax*, or the *main syntax*.

Alternatively, an *annotation syntax* is supported to add metadata to triples [15]. The annotation syntax is enclosed in curly braces and vertical bars ($\{ | \}$) and contains what is called a *PredicateObjectList* in the Turtle standard [14]. The *PredicateObjectList* is a sequence of (predicate, object) pairs that follows the annotated triple, using it as subject to create RDF-star triples. We illustrate with an example.

Example 3.6 The RDF document below, written in the Turtle-star syntax, describes the score of a movie and its source.

```
:data/TheGodfather1972 :hasScore "9,2".
<< :data/TheGodfather1972 :hasScore "9,2" >> :source "IMDB".
```

The same graph can be expressed using the annotation syntax:

```
:data/TheGodfather1972 :hasScore "9,2" { | :source "IMDB" | }.
```

In this case the quoted triple plays the role of subject in the `:source` relationship. As we want the triple to also be asserted we need to type out two lines using the main syntax, but only one line using the annotation syntax. In this example we have written in Turtle-star the RDF-star triple described in Example 3.3. ◇

Note that the annotation syntax can easily be converted to the main syntax, but as the triple being annotated always is asserted it is not possible to convert every graph written in the main syntax to the annotation syntax. Therefore we will focus

our discussion and examples on the main syntax. Another way to express this is to say that the less/greater than syntax is more expressive than the annotation syntax as it allows the use of a triple in the subject position without asserting it in the RDF-star graph.

Throughout this thesis, just as we've used Turtle before to express RDF-graphs, we will be using Turtle-star for RDF-star graphs, unless otherwise mentioned.

3.3.2 TriG-star, N-Triples-star, N-Quads-star

Additionally the community draft describes extensions of other popular concrete RDF syntaxes. As they are not relevant for this thesis we will not describe them further but refer the reader to the RDF-star draft report [15].

3.4 SPARQL-star

In addition to their extension of the RDF language, Hartig et al. have developed an extension to the SPARQL query language named SPARQL-star [15, 27]. Just like RDF-star builds on the concept of a triple, the SPARQL-star extension builds on the SPARQL concepts of triple pattern and basic graph pattern, see Definitions 2.5 and 2.6. The definitions below are sourced from Hartig [27] and the W3C community group working on RDF-star [15].

Definition 3.7 *SPARQL-star triple patterns* are defined recursively. First, every SPARQL triple pattern is also a SPARQL-star triple pattern. Second, let tp and tp' be SPARQL triple patterns, $\mathcal{I}$ the set of IRIs, $\mathcal{V}$ the set of SPARQL query variables, and $\mathcal{L}$ the set of RDF literals. Then (tp, p, o) , (s, p, tp') , and (tp, p, tp') are all *SPARQL-star triple patterns* [5, 27], under the assumption that

$$s \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}, \quad p \in \mathcal{V} \cup \mathcal{I}, \quad o \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}.$$

As you can see this definition closely follows the form of the definition of RDF-star triple. ◆

Note *Just like for standard RDF and SPARQL there is a disconnect here. SPARQL-star triple patterns admit literals in the subject position, unlike RDF-star triples where the subject is an IRI, blank node, or another RDF-star triple.*

Definition 3.8 A finite set of SPARQL-star triple patterns is a *SPARQL-star basic graph pattern* [15, 27]. ◆

The concrete syntax of the query language is extended in a similar way to how Turtle-star extends turtle. Either through the use of the double less/greater than signs, or through an annotation syntax. SPARQL-star is used to query an *RDF-star dataset*, which is one or more RDF-star graphs just like an RDF dataset is one or more RDF graphs.

Example 3.9 Recall the reified RDF data of Example 3.1, and the query retrieving films, their scores, and the source of that score. Assuming an equivalent RDF-star dataset, a SPARQL-star query to retrieve the same answers would look like this.

```
SELECT ?film ?score ?source
WHERE {
  ?film :hasScore ?score { | :source ?source | }.
}
```

Alternatively using the main syntax, the WHERE block is equivalently expressed as

```
WHERE {
  ?film :hasScore ?score.
  << ?film :hasScore ?score >> :source ?source.
}
```

Same as in Turtle-star, the annotation syntax is more concise. $\diamond$

In the example query both $(\text{?film :hasScore ?score.})$ and $(\text{<< ?film :hasScore ?score >> :source, ?source.})$ are SPARQL-star triple patterns. Taken together they form a SPARQL-star basic graph pattern.

Similarly to how they are defined for RDF-star triples, we define the *elements* of a triple pattern $tp = (s, p, o)$ as: $\text{Elmts}(tp) = \{s, p, o\} \cup \{\text{Elmts}(tp') \mid tp' \in \{s, o\} \cap \mathcal{TP}\}$, where $\mathcal{TP}$ is the set of all RDF-star triple patterns. For every RDF-star basic graph pattern the elements of that pattern is the union of the elements of each triple pattern tp in the graph pattern [27].

3.4.1 Algebra

Recall the definition of a SPARQL solution mapping from Definition 2.9. It has been extended by Hartig to allow for a formal semantics of SPARQL-star [27].

Definition 3.10 A *SPARQL-star solution mapping* is a partial function η from the set of query variables V to the set of RDF-star terms $\mathcal{T} \cup \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. Given a SPARQL-star basic graph pattern BGP , $\eta(BGP)$ is the basic graph pattern obtained by replacing the variables of BGP according to the mapping η . Call all variables for which η is defined the *domain* of η , denoted $\text{dom}(\eta)$ [27]. $\blacklozenge$

Definition 3.11 Let D be an RDF-star dataset, tp a triple pattern, and BGP an RDF-star basic graph pattern. The *evaluation* of a graph pattern over D , call it $\llbracket \cdot \rrbracket_D$, is a function from the set of RDF-star basic graph patterns that returns a set of SPARQL-star solution mappings. It is defined as [27]

$$\llbracket BGP \rrbracket_D = \{\eta \mid \text{dom}(\eta) = \text{Elmts}(BGP) \cup \mathcal{V} \text{ and } \eta(BGP) \subset T(G)\}.$$

These definitions allow for formal evaluation of SPARQL-star. $\blacklozenge$

3.4.2 Functions and operators

The SPARQL standard defines a number of functions and operators that apply to standard RDF terms and query variables allowing for example filtering of query results. SPARQL-star extends these to apply to RDF-star terms instead of only standard RDF terms. In addition, five new functions are introduced, three of which are modified functions from the SPARQL standard. For further information we refer the reader to the RDF-star community draft [15].

3.4.3 Answer formats

Depending on the dataset being queried, the SPARQL answer format might not be sufficient to capture variable bindings produced when querying RDF-star data.

Example 3.12 Consider the SPARQL-star query:

```
SELECT ?s
WHERE {<< ?s ?p1 ?o1 >> ?p2 ?o2}.
```

If it is evaluated over the RDF-star dataset consisting of the single triple: << :s :p1 :o1 >> :p2 :o2, there is no need for an extension of the SPARQL answer format. This is because there is only one answer, namely the solution mapping binding ?s to :s, and :s is an IRI, a RDF-term. If we instead execute the SPARQL query:

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

over the same dataset, the query variable ?s is bound to the quoted triple << :s :p1 :o1 >>. This triple is not an RDF term and is not supported as answer in the standard SPARQL answer formats. ◇

As the example above illustrates, there is a need to extend the SPARQL answer formats. This has been done by the RDF-star community group and the details can be read in the RDF-star draft [15]. The SPARQL JSON and XML answer formats have been extended to allow for quoted triples being bound to variables. This is a necessary addition to allow correctly querying RDF-star datasets [15].

4 Investigating RDF-star Support in VKG Systems

In this chapter we investigate which parts of a VKG system could be extended with RDF-star support, what the purpose of it is, and how it is done. We deduce that the mapping is the component most suitably extended and propose an extension to R2RML that we call R2RML-star. We detail how to extend VKG systems to support the relevant query language, SPARQL-star. In addition we discuss how the stages of query processing, meaning rewriting and unfolding, are affected by these changes, and finally there is a large example to illustrate the lessons of the chapter.

4.1 RDF-star Support in the Mapping

Recall that a VKG mapping, as defined in Definition 2.17, is a set of mapping assertions, each consisting of an RDF triple template and a SQL query. The triple template and SQL query are also known as the target and source part, respectively, of the assertion.

The mapping is a key component of a VKG system, it is used to populate the virtual knowledge graph with facts mapped from the data sources. For this reason we wish to extend the target part of the mapping assertion to support RDF-star triples.

Definition 4.1 We define a *triple-star template* $T_*(Y)$ as follows. First, every triple template, see Definition 2.16, is also a triple-star template. Second, assume that

$$T_*(Y) = (\text{subject}(Y_1), \text{predicate}(Y_2), \text{object}(Y_3)), \\ T'_*(Y') \text{ and } T''_*(Y'')$$

are all triple-star templates, then

$$(T'_*(Y'), \text{predicate}(Y_2), \text{object}(Y_3)) \\ (\text{subject}(Y_1), \text{predicate}(Y_2), T''_*(Y'')) \\ (T'_*(Y'), \text{predicate}(Y_2), T''_*(Y'')).$$

are also triple-star templates. ◆

Definition 4.2 An *RDF-star mapping* is a set of *mapping assertions*, each having the form: $[\mathcal{T}_*(Y) \leftarrow (Y = \text{sql}(\cdot))]$, where $\text{sql}(\cdot)$ is a SQL query over a data source

schema S , Y is the set of output variables of the SQL query, and $\mathcal{T}_*(Y)$ is a triple-star template. ◆

4.1.1 R2RML extension

In order to extend the concept of a mapping to that of an RDF-star mapping we propose a standard-breaking extension to the mapping language R2RML, for the purpose of this thesis we call it R2RML-star. This is done through the use of the prefix **star**:¹, a new R2RML term type representing RDF-star triples (thus breaking the standard), and a set of predicates specifying the subject, predicate, and object of the triple.

Some work has already been done to create a language mapping relational data to RDF-star by Delva et al. in the form of RML-star [28]. The reason for us choosing a different approach is that the RML mapping language was designed for Extract-Transform-Load (ETL) use and is not suitable for creating virtual knowledge graphs. RML is an extension of R2RML supporting unstructured forms of data such as XML and CSV files [16]. These files are often slow to access and parse as they don't support for example indexing, making accessing them through a VKG approach unsuitable.

The R2RML language specifies a language to describe mappings between relational databases and RDF graphs, it is described in Section 2.6.1. Part of this mechanism lies in the term maps of R2RML, which specify how to generate RDF terms [9]. We extend this mechanism to have the ability to generate RDF-star terms.

Every column- and template-valued term map has an inferred or specified term type which specifies which type of RDF term is generated. In standard R2RML these are: IRI, blank node, and literal. To add support for the additional RDF-star term of quoted triple, we define a new term type: **star:RDFStarTermType**, which in R2RML-star can be the specified term type in subject or object maps. This breaks the standard and leads to any software supporting standard R2RML and not our extension to produce an error, which however is a desirable property.

In order to specify the subject, predicate, and object of the quoted triple to be generated, three new properties are defined: **star:subject**, **star:predicate**, and **star:object**. The object of the **star:subject** and **star:object** properties must be an object map, which in turn can have any term type, including the new RDF-star term type representing a deeper nested triple. The **star:predicate** property takes a predicate map as object and it can have only one term type: IRI.

Example 4.3 To see the full mapping featured in this example see Mappings 2 and 3 in the appendix. The triple-star template $[(\texttt{/}\{\texttt{Name}\}\{\texttt{Year}\}, \texttt{rdf:type}, \texttt{:Film}), \texttt{:source}, \texttt{"IMDB database"}]$ would be expressed as follows in R2RML-star.

```
rr:subjectMap [ a rr:SubjectMap;
                rr:termType star:RDFStarTermType;
```

¹star being the IRI: <https://w3id.org/obda/r2rmlstar#>


```

star:subject [a rr:ObjectMap;
  rr:template "http://lukas.thesis.org/films/{Name}{Year}";
];
star:predicate [a rr:PredicateMap;
  rr:constant rdf:type
];
star:object [a rr:ObjectMap;
  rr:constant :Film
];
];
rr:predicateObjectMap [
  rr:predicate :source;
  rr:object "IMDB database";
].

```

The triple-star template `[:/ {Name} {Year}, :isMentionedBy, (:/ {Name} {Year}, :releasedIn, "{year}")]` would be expressed as follows in R2RML-star.

```

rr:subjectMap [ a rr:SubjectMap;
  rr:template "http://lukas.thesis.org/films/{Name}{Year}";
];
rr:predicateObjectMap [
  rr:predicate :isMentionedBy;
  rr:objectMap [ a rr:ObjectMap;
    rr:termType star:RDFStarTermType;
    star:subject [a rr:ObjectMap;
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      rr:termType rr:IRI;
    ];
    star:predicate [a rr:PredicateMap;
      rr:constant :releasedIn;
    ];
    star:object [a rr:ObjectMap;
      rr:column "year";
      rr:termType rr:Literal;
    ];
  ];
].

```

These are simple examples to illustrate the basics of R2RML-star but more complex templates are of course possible. As follows from the definition of `RDFS-starTermType`, nesting is also allowed, just as in RDF-star. ◇

4.2 SPARQL-star Support

A part of implementing RDF-star support is to support the extended query language SPARQL-star. In order to do this the VKG system's query processing must be extended from supporting the normal SPARQL algebra described in Section 2.3.3, to supporting the extended SPARQL-star algebra of Section 3.4.1

Recall that a VKG system goes through several stages of query processing. After parsing the query, it is manipulated through the stages of rewriting and unfolding with regards to the ontology and mapping respectively. Both of these processes must be extended to take into account the new grammar of the SPARQL-star algebra.

4.2.1 Parsing SPARQL-star queries

The first step to SPARQL-star support is the correct parsing of SPARQL-star queries. This means that the VKG system must correctly read a SPARQL-star query; the system must be able to recognize SPARQL-star triple patterns, SPARQL-star functions and operators, and other aspects of the SPARQL-star standard as described in Section 3.4.

4.2.2 Extending query rewriting

VKG systems rewrite posed queries with the help of the supplied ontology in order to enhance the answers to the query, we call this *query rewriting*. This is done through the process of SPARQL entailment, see Section 2.5.3. In order to support entailment under SPARQL-star, or standard SPARQL queries combined with an RDF-star mapping, the VKG system needs to be extended to apply entailment rules to RDF-star triples.

Example 4.4 Consider the RDF-star dataset consisting of the single triple `<< :John a :Actor >> :source "IMDB"`. and an ontology with the single statement `:source rdfs:range :Database`. In this case the query

```
SELECT ?person
WHERE {
  << ?person a :Actor >> :source ?source.
  ?source a :Database.
}
```

should return `:John` as only answer, if the system runs under an RDFS entailment regime. $\diamond$

We call this form of rewriting simple rewriting. That is, when the ontological rule applies to the root RDF-star triple and not to a quoted triple.

Definition 4.5 *Simple rewriting* is the process of query rewriting when SPARQL entailment rules are applied to root RDF-star triples. That is, for each asserted ontological rule ϕ and RDF-star triple t_1 such that $\phi(t_1) \implies t_2$, the VKG system

handles simple rewriting if a query that returns a variable binding to t_1 , also returns a binding to the same variable, t_2 . ♦

Example 4.6 Consider the RDF-star dataset consisting of the single triple `<< :John a :Actor >> :source "IMDB"`. and an ontology with the single statement `:Actor rdfs:subclassOf :Person`. In this case the query

```
SELECT ?person
WHERE {<< ?person a :Person >> :source "IMDB".}
```

should return `:John` as only answer, if the system runs under an RDFS entailment regime. ◇

Here the entailment has to be applied to a quoted triple which makes the rewriting more complex and we thus refer to this case as complex rewriting.

Definition 4.7 *Complex rewriting* is the process of query rewriting when SPARQL entailment rules are applied to quoted triples, and possibly root RDF-star triples. For each asserted ontological rule ϕ and RDF-star triple t_1 such that $\phi(t_1) \implies t_2$, the VKG system handles complex rewriting if a query that returns a variable binding to (t_1, p, o) , also returns a binding to the same variable, (t_2, p, o) . Similarly the rewriting should apply also if the triple takes the place of object in an RDF-star triple. ♦

In order for a VKG system to have full RDF-star support at least simple rewriting should be supported. As for complex rewriting, the RDF-star community is still undecided on how it should be implemented. Rewriting quoted triples can lead to undesirable results, as the following example illustrates.

Example 4.8 Consider the ontology consisting of the single RDFS statement `(:Actor, rdfs:subClassOf, :Person.)`. This is combined with the RDF-star graph consisting of a single RDF-star triple:

```
<< :John, rdf:type, :Actor >>, :dateAdded, "2022-03-03".
```

The quoted triple denotes John being an actor, and the annotation specifies when the triple was added. Now when a VKG system is running and rewriting this triple with regard to the ontology a new triple will be implicitly generated:

```
<< :John, rdf:type, :Person >>, :dateAdded, "2022-03-03".
```

Unless the system is running on the third of March 2022, this triple is factually incorrect. Similar issues arise in many use cases where the annotation of a triple does not necessarily apply to the output of its rewriting. ◇

4.2.3 Extending query unfolding

Once a query has been parsed and rewritten, it is necessary to unfold it with regards to the mapping. The VKG system must be extended to handle the new triple patterns, functions, and operators as described in Section 3.4.

Even if we don't include these new parts of the grammar from the query, so that it looks like a standard SPARQL query, the unfolding must be extended. This is because querying RDF-star data might return answer variables bound to quoted triples, regardless of the fact that a standard SPARQL query does not contain SPARQL-star triple patterns.

Example 4.9 Consider the RDF-star dataset consisting of the single triple from Example 4.4 (`<< :John a :Actor >> :source "IMDB".`), the query

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

returns a single value bound to `?s`, the quoted triple: `:John a :Actor`. $\diamond$

R2RML-star unfolding

As we know, VKG systems do not query RDF datasets directly. In Example 4.9 above, the triple retrieved must have been generated from a mapping. To our knowledge the only mapping that generates RDF-star data is our proposed R2RML-star extension. In this section we describe how a VKG system handles R2RML-star.

The query unfolding in a VKG system fundamentally works through the process of graph matching. The SPARQL query's WHERE clause is matched with the target part of the mapping. Already here R2RML-star poses problems as each quoted RDF-star triple can match with either one or three query variables.

Example 4.10 A mapping with the target part (`[:{Person}, rdf:type, :Actor], :source, "IMDB"`), where `{Person}` refers to a database column, should match to both of the queries below.

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

```
SELECT ?s
WHERE {<<?s ?p1 ?o1>> ?p2 ?o2.}
```

Assuming that this database column contains "John" as its only value, each query should return one answer each, `<< :John a :Actor >>` and `:John`, respectively. $\diamond$

The VKG system must be extended to correctly handle these two scenarios, which is non trivial. The system must also be extended to correctly handle the accompanying source part of the mapping, the SQL queries, and execute them over the data sources.

4.2.4 Extending query answers

As shown by Example 4.9 there is a need to extend the answers returned by the VKG system. Quoted triples can be bound to variables and the system must be changed to take this into account and produce as output the SPARQL-star standard's new answer formats as described in Section 2.3.2.

4.2.5 Adding support for new functions and operators

In Section 3.4.2 we saw that SPARQL-star redefines the standard SPARQL functions and operators, and introduces new ones. This must be taken into account when extending a VKG system with SPARQL-star support and the internal query processing must be extended to correctly apply the updated and new functions and operators.

4.3 End to End Example

We showcase here a complete end to end example demonstrating the power of a VKG system extended with RDF-star support. As a first step we introduce a data integration scenario. The next subsection focuses on how it would be solved with a VKG system supporting only standard RDF. Finally we show an example of how a RDF-star system as described in this thesis could operate.

4.3.1 The data integration scenario

Recall the two databases of Example 2.12, visually described in Tables 2.1 and 2.2. The data stored is sourced from two different popular websites, the Internet Movie Database (IMDB) and Rotten Tomatoes. For each database, the data contains three pieces of information, the film's name, release year, and score as given on the source website. An example entry of the IMDB database looks as such:

Name: Pulp Fiction; Year: 1972; Score: 9.2.

Now, our hypothetical end user is interested in integrating these two data sources. That is, they want to combine them into one for the purpose of querying them simultaneously. There are many ways to this but let us assume that he or she wants to integrate them using the VKG approach. A particular problem arises due to the heterogeneity of the data sources. The scores of the movies differ between the two sources. Not only the scores themselves but also the format, how should our user solve this conundrum?

We assume that the user wants to preserve the scores as they are, without converting them to the same format for example. But some information should be added, such as meta-data specifying from which data source the score comes.

4.3.2 A standard RDF solution

We will focus on the mapping of the VKG solution as that is the crucial part.

Example 4.11 To map the score in R2RML we use the following two mappings, written in a simplified syntax:

```
:films/{Name}{Year} :score {Score}.
  <- SELECT Name, Year, Score FROM imdbTable1
:films/{MovieName}{ReleaseYear} :score {Rating}.
  <- SELECT MovieName, ReleaseYear, Rating FROM RtTable1
```

The first mapping expressed in R2RML is:

```
<#Mapping-film-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
  ];
  rr:predicateObjectMap [
    rr:predicate :score;
    rr:objectMap [ rr:column "Score" ];
  ].
```

In order to include the desired metadata, we make use of RDF reification. This leads us to this far more complex mapping:

```
<#Mapping-reify-annotate-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "Statement{Name}{Year}{Score}";
    rr:termType rr:BlankNode;
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:subject;
    rr:objectMap [
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      rr:termType rr:IRI; ];
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:predicate;
    rr:objectMap [
      rr:constant :score ; ];
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:object;
    rr:objectMap [
      rr:column {Score}; ];
  ];
  rr:predicateObjectMap [
    rr:predicate :source;
    rr:objectMap [
      rr:constant "IMDB" ];
  ].
```

The usage of RDF reification in this way is necessary in order to annotate triples, however it adds some difficulty for the end user of the system. For example, this

intuitively correct SPARQL query asking for all film scores will not return any answers using the reification based mappings.

```
SELECT ?film ?score
WHERE { ?film :score ?score }
```

Instead the more complex query below will give the expected results.

```
SELECT ?film ?score
WHERE {
  ?statement rdf:subject ?film .
  ?statement rdf:predicate :score .
  ?statement rdf:object ?score . }
```

This complicates the use of the system for the end user, not to mention the performance impact on the system when it has to handle more complicated queries. ◇

4.3.3 A R2RML-star mapping

Next, we wish to simplify by using our proposed R2RML-star extension.

Example 4.12 The metadata infused mapping of the previous section can be rewritten in R2RML-star as shown below.

```
<#Mapping-film-score>
rr:logicalTable [ rr:tableName "imdbTable1" ];
rr:subjectMap [
  rr:template "http://lukas.thesis.org/films/{Name}{Year}";
];
rr:predicateObjectMap [
  rr:predicate :score;
  rr:objectMap [ rr:column "Score" ];
].

<#Mapping-star-annotate-score>
rr:logicalTable [ rr:tableName "imdbTable1" ];
rr:subjectMap [
  rr:termType star:RDFStarTermType;
  star:subject [a rr:ObjectMap;
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
  ];
  star:predicate [a rr:PredicateMap;
    rr:constant :score
  ];
  star:object [a rr:ObjectMap;
    rr:column {Score};
```

```

];
];
rr:predicateObjectMap [
  rr:predicate :source ;
  rr:objectMap [ rr:constant "IMDB" ];
].

```

The first mapping here, `<#Mapping-film-score>` was used also in the previous section. It uses standard RDF and serves to assert the triple in the graph. The second mapping has the same triple pattern but generates quoted triples, along with the annotated `:source`.

Using this R2RML-star mapping, the system remains intuitive for the end-user and the following SPARQL query will return the expected results.

```

SELECT ?film ?score
WHERE { ?film :score ?score }

```

In addition one can easily retrieve the associated metadata with this SPARQL-star query:

```

SELECT ?film ?score ?source
WHERE { << ?film :score ?score >> :source ?source .}

```

This simplicity stands in stark contrast to the complexity of RDF reification. ◇

5 Implementing RDF-star Support for Ontop

In this chapter we apply the studies done in the previous chapter to extend the open source system Ontop with RDF-star support. We describe what needs to be implemented, how it is done, and also the parts of Ontop’s internal functioning that are necessary to understand the other descriptions. The functions that are extended include query parsing, rewriting, and unfolding, and the parsing and handling of R2RML-star mappings. We also describe work that is in progress, including the support of SPARQL-star answer formats and SPARQL-star functions and operators. We explain internal Ontop structures such as the Intermediate Query and query nodes.

5.1 Parsing the Query

As described in Section 4.2.1 a VKG system must be able to parse SPARQL-star queries in order to support them, and we show now how this support has been added to Ontop. Ontop uses the RDF4J library to parse SPARQL queries input by the end user [22]. RDF4J is a popular Java library used to interact with RDF data, it includes support for reading and parsing not only the standard concrete RDF syntaxes, but also RDF-star data, SPARQL queries, and SPARQL-star queries [25]. Hence, by virtue of RDF4J’s SPARQL-star support, Ontop supports the parsing of SPARQL-star queries.

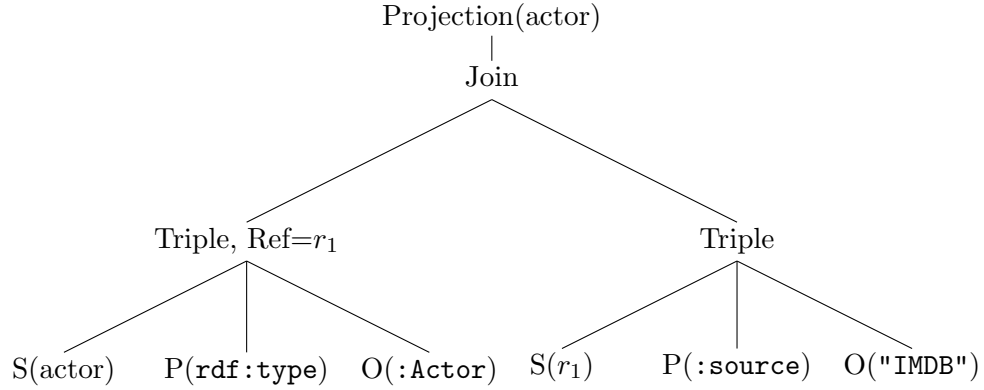
5.1.1 RDF4J’s parsing of SPARQL-star queries

SPARQL and SPARQL-star queries are parsed by RDF4J into a tree structure, call it an *RDF4J Tree*, representing the SPARQL algebra. The trees contain nodes representing operations of the SPARQL algebra. For SPARQL-star queries a design decision has been taken wherein an RDF-star triple containing quoted triples is not represented by a single node, but rather by a sub-tree of joins, triples, and quoted triples. Each quoted triple is represented as a triple with an added fourth element used as reference. A pointer to this reference then takes the place of subject or object in the encompassing RDF-star triple.

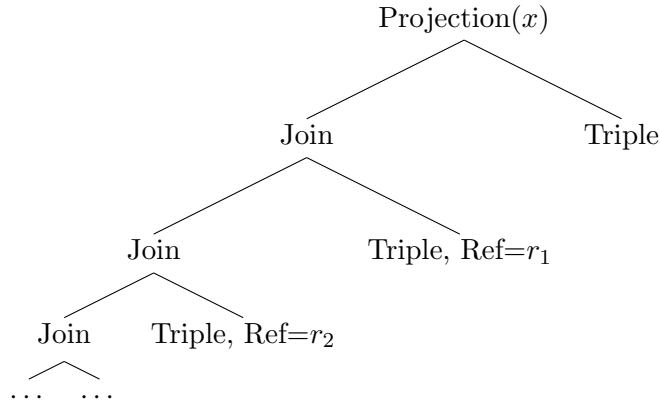
Example 5.1 The SPARQL query

```
SELECT ?actor
WHERE { <<?actor rdf:type :Actor>> :source "IMDB". }
```

is parsed into the following RDF4J tree



Where S, P, O signify the subject, predicate, and object respectively of the RDF-star triple. A deeply nested triple generates multiple joins as RDF4J's join node is a binary operator. For example the generated RDF4J tree of a deeply nested triple might look like this:



This limitation on RDF4J's join node leads to convoluted trees coming from SPARQL-star queries. We will later show how this issue is dealt with when it comes to Ontop's internal data structures. $\diamond$

5.2 Intermediate Query

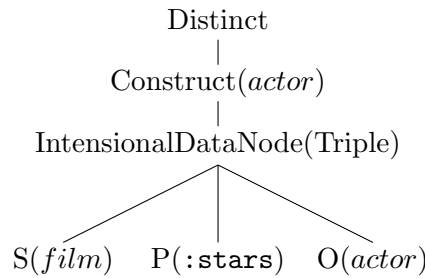
In order to bridge the gap between SPARQL queries on the one hand and SQL queries on the other hand, the VKG system Ontop uses a data structure called the *Intermediate Query*, or *IQ* for short. This is the format that is used to store a query throughout all the stages of query manipulation that Ontop performs. This includes rewriting the query in regard to the ontology, unfolding the query in regards to the mapping, and optimizing the query. The SPARQL query posed by the system's end user is first turned into an RDF4J tree independent of Ontop, then directly transformed to an IQ before these stages. At the end of the manipulation most of the IQ is converted to SQL queries to be evaluated over the data sources by the external Database Management Systems [3].

Through the work of this thesis we have extended many parts of Ontop that interact with this data model and to understand these changes it is necessary to understand the IQ itself. The IQ is at its core a tree structure built up of different *Query Nodes* that we describe further below.

Example 5.2 The SPARQL query

```
SELECT DISTINCT ?actor
WHERE {
  ?film :stars ?actor .
}
```

Is parsed by RDF4J to construct an RDF4J tree. That tree is then converted by Ontop into the IQ:



The distinct node is self explanatory, the construct node signifies projection, which variables are projected upwards through the tree, and the data node will be explained below. ◇

5.2.1 Query nodes

The Query nodes are an abstraction representing query operations common to both the SPARQL and SQL query languages, for example the distinct node which eliminates duplicate answers. The design of these nodes is complex as the two query languages have their differences and the IQ must be sufficiently abstract to represent them both. We will present those parts of the IQ that are relevant for our work. For a full characterization we refer the reader to Ontop’s documentation for developers [23, 24].

Data nodes and predicates

Data nodes are the typical leaf nodes of the IQ. They represent the data that will ultimately be returned to the system’s end user. They come in two kinds, *intensional* and *extensional* data nodes. Intensional data nodes represent data in the RDF format such as triples and quads. During query unfolding, these nodes are transformed by Ontop into extensional data nodes containing the information necessary to execute SQL queries over the source databases. Our work does not touch on extensional data nodes and we will focus our efforts on describing intensional nodes.

Each intensional data node contains a *predicate* acting as a marker for its type. The version of Ontop supporting only standard RDF contains two such predicates that could be contained in an intensional data node, triple and quad. We have extended this to a hierarchy of predicates capable of signaling three properties necessary to represent RDF-star triples: does the triple have a quoted subject, does it have a quoted object, and is it itself a quoted triple? An example of a predicate combining answers to these questions is the TripleRefNestedSubject predicate. It has a quoted subject, but no quoted object, and the Ref denotes that it is itself a quoted triple. The full list of such predicates is eight long and of course includes also the standard RDF triple, which is not itself quoted and contains no quoted triples. The full list:

- Triple
- TripleNestedSubject
- TripleNestedObject
- TripleNestedSO (Nested Subject & Object)
- TripleRef
- TripleRefNestedSubject
- TripleRefNestedObject
- TripleRefNestedSO

These nodes and their predicates first appear during runtime as the system translates the end user's SPARQL-star query into an IQ. The triple patterns of the SPARQL query are internally represented by intensional data nodes. For a SPARQL-star query, any triple-star patterns are represented by data nodes containing the newly created predicates. The next time they appear is when an R2RML-star mapping is read by Ontop. The target part of the mapping is translated into its own IQ tree, to be matched with a part of or the whole IQ tree coming from the SPARQL query.

This typing system identifying RDF-star triples as separate from those of standard RDF, along with their properties, gives the system a tool by which to avoid incorrectly matching mappings with the wrong level of nesting with a SPARQL-star query. It could also be used to implement many of the new SPARQL-star operators as described in Section 3.4.2.

In addition to the predicate, each data node contains data. In the case of intensional data nodes, regardless of predicate, they represent an RDF(-star) triple, hence they contain three data points: subject, predicate, and object.

Example 5.3 A simple RDF triple such as (`:John`, `rdf:type`, `:Actor`) is represented by the intensional data node:

IntensionalDataNode(Triple)

S(:John) P(rdf:type) O(:Actor)

Here the node's predicate is denoted within parentheses (Triple), and the node's data shown by the subject, predicate, object triple below. ◇

Example 5.4 More common in VKG systems is that an intensional data node represents the target part of a mapping. The R2RML-star mapping

```
((<:{Person}, rdf:type, :Actor>>, :source "Table1") <-
  SELECT Person FROM Table1
```

is read in and represented by the IQ:

Join

IntensionalDataNode
(TripleRefSimple[_ : b1])

IntensionalDataNode
(TripleNestedSubject)

S(:{Person}) P(rdf:type) O(:Actor) S(_ : b1) P(:source) O("Table1")

As you can see one data node is generated for each triple-star pattern occurring in the query. ◇

5.3 Rewriting SPARQL-star Queries in Ontop

We have extended Ontop's rewriting capabilities to apply not only to standard RDF triples, but to RDF-star triples as well. This was accomplished through allowing rewriting to be applied to the new data node types, those with predicates signifying RDF-star triples. We have elected to apply only simple rewriting, not complex rewriting, see Definitions 4.5 and 4.7. This is due to the current indecision of the community on if complex rewriting should be supported. See Example 4.8 for an illustration of the issues surrounding complex rewriting.

5.4 Unfolding SPARQL-star Queries in Ontop

In section 4.2.3 we describe to what extent a VKG system must be expanded to support the unfolding of SPARQL-star queries. In Ontop the query unfolding process consists of two steps. First the SPARQL query must match with the relevant mapping, in a correct way such that the right mappings match with the right query.

This is done through a process known as pattern matching. In practice in Ontop it works by checking that the query's IQ representation, or a sub-tree of the IQ, matches with the IQ representing the target part of the mapping, and then replacing that sub tree by the mapping's IQ. We delve into further detail on this in Section 5.4.2. This will populate the IQ being processed with RDF terms.

The second step is the conversion of intensional data nodes into extensional ones. All RDF terms and functions are replaced with data to be retrieved from the relational data sources and database functions before these queries are sent off to their respective DBMS. This is done by replacing the parts of the IQ being processed that are identical to the mappings' target parts with the relevant mappings' source parts.

5.4.1 Motivation for matching

Let us focus our discussion on the matching process and take the easiest example first: a standard SPARQL query, free of SPARQL-star grammar. Recall the typical *s p o* query, a SPARQL query whose **WHERE** clause is just $\{s \ p \ o\}$. Such a query will match with every triple of the queried dataset. As the query is free from RDF-star specific grammar it will be treated as any other standard SPARQL query during the first step of processing: parsing. The IQ arising will have one intensional data node: $\text{triple}(s, p, o)$.

However, just because such a query is free from SPARQL-star grammar does not mean that it is not affected by our work. If the data queried contains quoted triples, in Ontop meaning the mapping is an R2RML-star mapping populating the virtual Knowledge Graph with quoted triples, then quoted triples should be bound to the answer variables. This means we must be able to match such standard SPARQL triples with R2RML-star mappings without changing the behaviour of the query when combined with standard R2RML mappings.

In the more complex case, the query posed is a SPARQL-star query having for example a quoted triple in the subject of a triple pattern. In this case the query should be matched with a suitable R2RML-star mapping and must not match with any standard R2RML mappings.

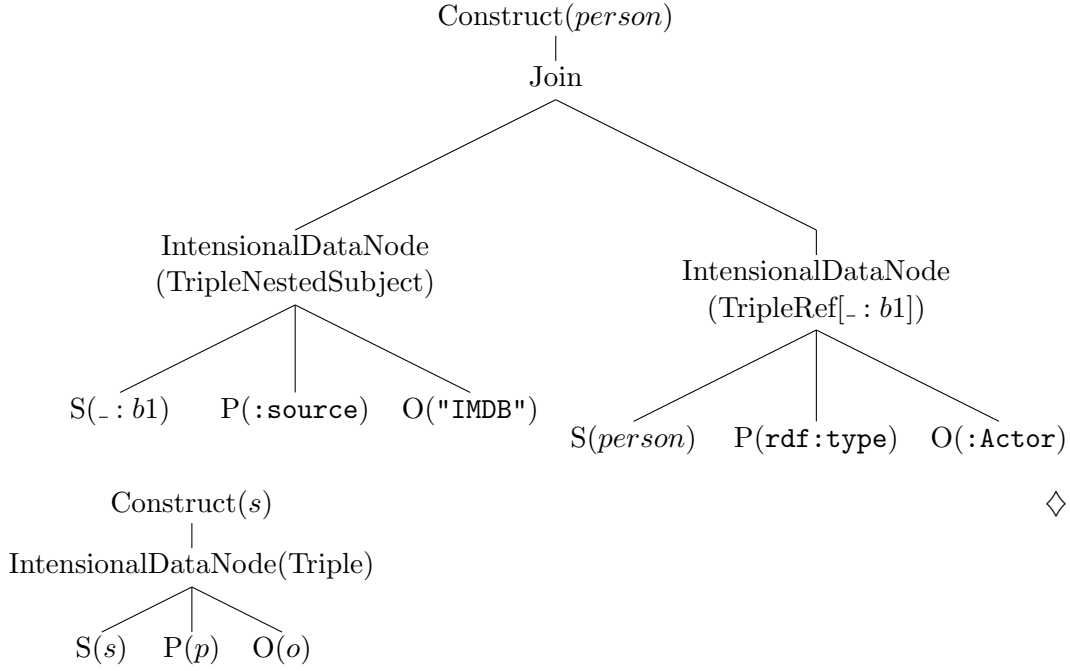
Example 5.5 In this example we use similar queries as in Examples 4.4, 4.6, and 4.9 from the previous chapter. Having a dataset consisting of the single triple $\langle\langle :John \ a \ :Actor \ \rangle\rangle \ :source \ "IMDB"$, we wish to run two SPARQL queries on this dataset. The SPARQL-star query:

```
SELECT ?person
WHERE {<< ?person a :Actor >> :source "IMDB".}
```

and the standard SPARQL query:

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

They are read by Ontop and converted to the two IQs respectively:



5.4.2 Translation from R2RML-star to IQ

To discuss pattern matching of SPARQL-star queries with R2RML-star mappings we must first describe how such mappings are parsed and read. In Section 4.1.1 we detail how we extend R2RML and in this section we will elaborate on how Ontop deals with R2RML-mappings.

For each R2RML mapping Ontop generates a set of internal Ontop specific mapping objects, call them *i-mappings*, for internal mapping. All the *i-mappings* generated from one R2RML-mapping have the same source part corresponding to the source part of the R2RML-mapping. The source part of the *i-mapping* is an IQ consisting of an extensional data node, representing the R2RML mapping's SQL query. As this thesis does not touch on the source part we will not discuss it further.

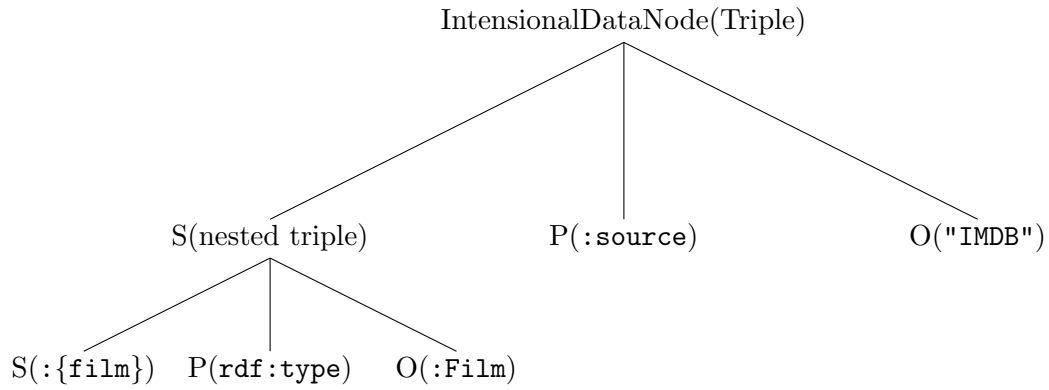
A single R2RML mapping can contain multiple triple or quad templates and the purpose of Ontop's *i-mapping* is to divide the R2RML mapping into smaller pieces where each piece, each *i-mapping*, contains one triple template in its target part. This also make them easier to process.

In the initial stage of query parsing an R2RML-star mapping is parsed the same way as a standard R2RML mapping. An IQ consisting of one intensional data node is created to be the target part of each *i-mapping*, having the predicate of triple. Should the term type of a subject or object be the RDF-star term type, the system knows that the mapping writer is using a quoted triple in that position.

The system will then read the specific RDF-star properties of subject, predicate, and object in order to populate the intensional data nodes with the quoted triple. This is done by essentially wrapping the three RDF-star terms just read in, to a specific object, a nested triple function, that fits in Ontop's typing system and can take the place of subject or object in the root triple. This process is recursive and multiple levels of nesting are possible as defined by the R2RML-star extension, see Section 4.1.1. This is compatible with recursively nested triples as specified by the RDF-star draft [15].

At this point the i-mapping can already match with a standard SPARQL query as in the example below.

Example 5.6 Consider Mapping 2 from the Appendix, we express its target part simplified here as: $([:\{\text{film}\}], \text{rdf:type}, :\text{Film}], :\text{source}, \text{"IMDB"})$. When Ontop parses the mapping this is initially parsed just as any standard R2RML mapping, generating the IQ:



Due to the triple predicate in the intensional data node this matches to any SPARQL query that has a triple with query variables in the WHERE clause, e.g. asking for $\{?s ?p ?o.\}$. This would then introduce a quoted triple bound to the query variable $?s$. This IQ would not match to a SPARQL-star query explicitly asking for quoted triples, e.g. a WHERE clause containing $\{<< ?s ?p1 ?o1 >> ?p2 ?o2.\}$, as this query would be parsed into an entirely different IQ and the process of pattern matching fails. $\diamond$

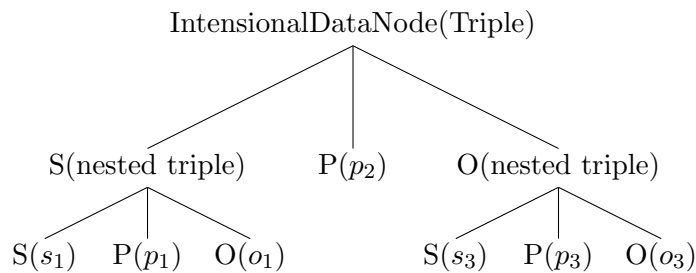
Extending the translation

In order to match R2RML-star mappings with SPARQL-star queries this translation process needs to be extended further. For each i-mapping containing an intensional data node that has a quoted triple in the subject or object position, a new i-mapping with the same source part and a new target part is created. This creation of a new target IQ follows the following algorithm:

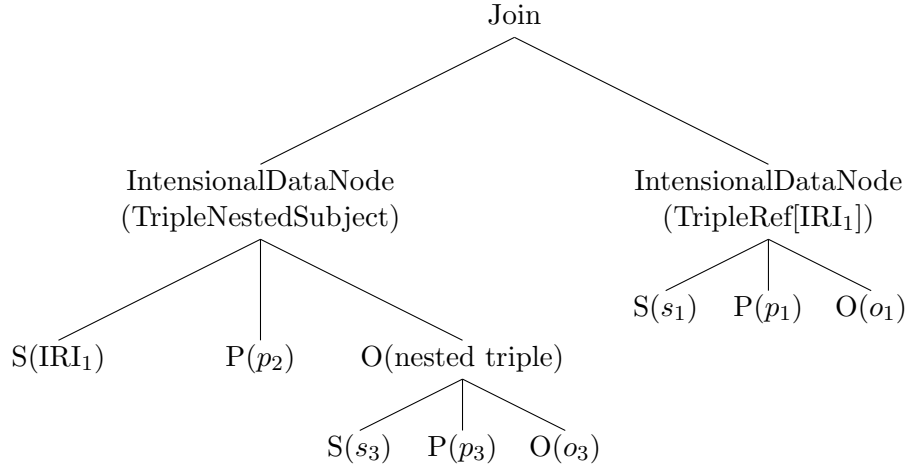
1. Check if an intensional data node in the IQ representing the i-mapping's target part has a nested triple function in the subject position, if yes proceed to Step 2, otherwise to Step 5.
2. Use the subject, predicate, and object of the nested triple function to create a new intensional data node with the simple tripleRef predicate, give it a unique IRI as reference.
3. Replace the nested triple function with the unique IRI created in the previous step. Change the original node's predicate to add the property of having a nested subject.
4. Replace the original intensional node's position in the IQ with a new tree consisting of a JOIN as parent, and the two newly created nodes below.
5. If no nested triple function was found in Step 1, look for a nested triple function in the object position. Proceed in a similar manner as Steps 2-4 did but now considering the object as nested.

Note that this algorithm is applied to each i-mapping and can thus be considered recursive as it is applied even to i-mappings already generated by the algorithm. This means that it also handles RDF-star triples that contain quoted triples in both the subject and object positions.

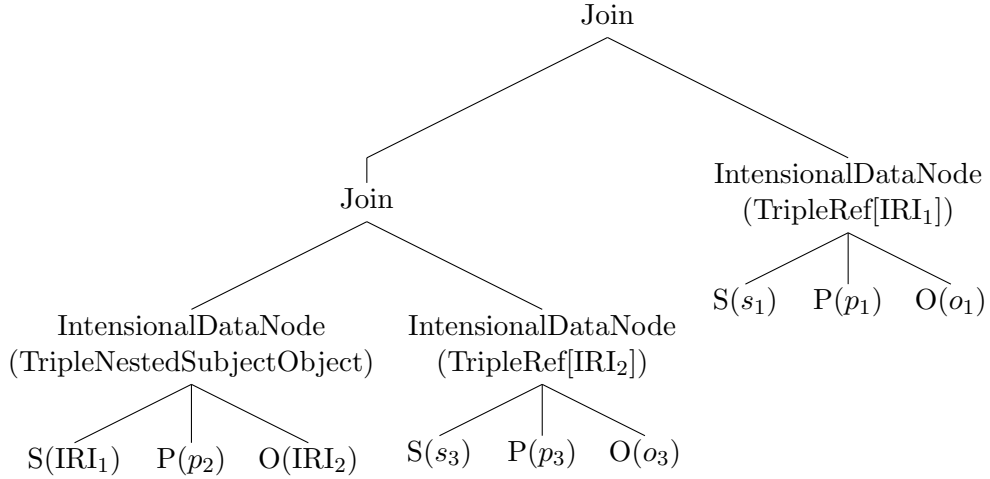
Example 5.7 Consider the i-mapping target represented by the following IQ:



Such an IQ will already match with IQs generated by standard SPARQL queries asking for triples. It will however not match with any SPARQL-star queries specifically asking for nested triples. When the algorithm described above is applied to this IQ the result is a new IQ:



At this point the IQ will match sub trees of IQs representing SPARQL-star queries containing a nested triple in the subject position in the where clause. A further pass of the algorithm produces the following IQ:



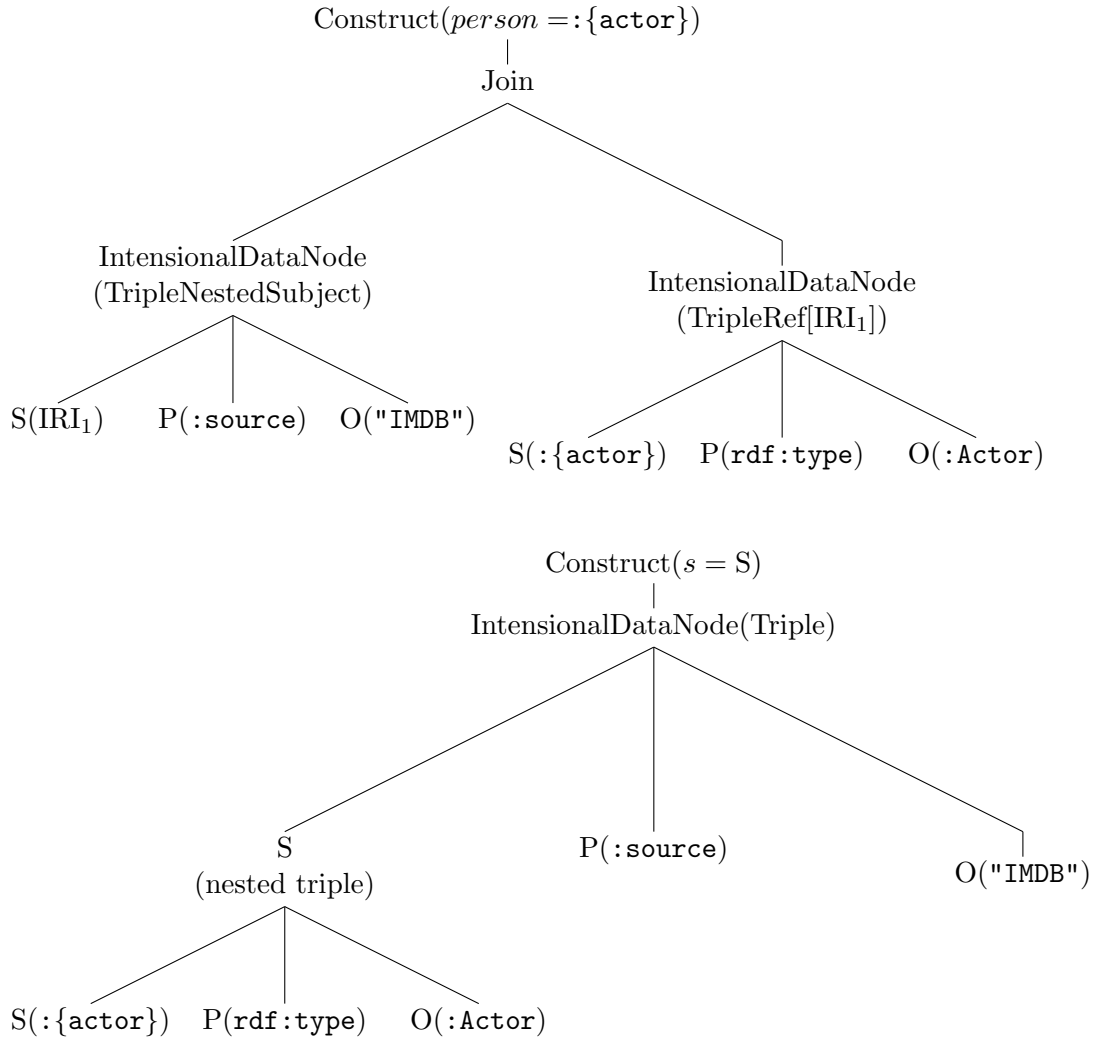
One can imagine that for a deeply nested structure a lot of join nodes are produced. Luckily the IQ's join query node is n-ary and can have more than two children. At a later stage of query optimization such structures are simplified and every data node is lifted up to sit under one parent join node. ◇

5.4.3 Conversion from intensional to extensional data node

After the query's IQ has been matched with the IQ's of each relevant mapping, the unfolding process must continue in order to execute SQL queries over the data sources. All RDF terms, functions, and operators must be converted to database terms and functions or otherwise lifted up to the top of the query so that the DBMS does not get passed any RDF terms.

Ontop already supports this process but through our work we have introduced a new term, namely the quoted triple, this is internally represented through a nested triple function. At this time we have been unable to implement support for translating this function in Ontop. This means that only a limited amount of query and mapping combinations are supported at this time, namely those queries and mappings that match without introducing the nested triple function to the query's IQ.

Example 5.8 Consider a VKG instance with only one mapping having the target part: $([:\{\text{actor}\}, \text{rdf:type}, :\text{Actor}], :\text{source}, \text{"IMDB"})$. Wishing to run the two queries from Example 5.5, they will both match with the mapping according to the process we have outlined above. halfway through the unfolding process we end up with the following two IQs:



Of these queries the first one will pass through the second stage of unfolding, execute correctly over the data sources, and return the expected answers. The second query will fail due to the current inability of Ontop to translate the nested triple function. $\diamond$

5.5 Supporting SPARQL-star Answers in Ontop

In Section 4.2.4 we described how VKG systems should support additional query answers under SPARQL-star. Our answer to this question ties to Section 5.4 on how Ontop unfolds SPARQL-star queries and translates R2RML-star mappings. In order to support binding quoted triples to SPARQL answer variables there must be support for the nested triple function at the latter stage of unfolding. As we have not yet reached that milestone, Ontop at the moment only supports returning standard answers to SPARQL ASK and SELECT queries, with standard RDF terms bound to the answer variables.

Similarly there has been no work done to implement SPARQL-star specific answers to the other types of query, ASK, DESCRIBE, or CONSTRUCT, and they have limited functionality. DESCRIBE and CONSTRUCT can not create RDF-star graphs under Ontop and ASK only functions under a VKG environment that does not give rise to the nested triple function as described in the previous section.

5.6 Supporting SPARQL-star Functions and Operators

Section 4.2.5 describes how standard SPARQL functions and operators must be extended under SPARQL-star, and newly defined operators and functions. Due to time constraints we have not been able to implement this functionality in Ontop during the work of this thesis, although we have laid some groundwork.

The extensions necessary fall in two categories. Redefining the standard SPARQL functions is one of them, and for those the domain must be extended to RDF-star terms and their behaviour should be defined for such terms. For example, the BOUND function should return true if a query variable is bound to an RDF-star term such as a quoted triple [15].

The second category contains new functions such as isTRIPLE which takes in one RDF-star term and returns a boolean indicating if the term is a triple or not [15]. For these functions the groundwork has been laid in the typing system outlined in Sections 5.2.1 and 5.4.2. In the specific example one can check if the term as it is represented in the IQ is either the nested triple function or holds a reference to an intensional data node with a type of tripleRef predicate.

6 Conclusion

This chapter forms the final part of the thesis. In it we summarise the results of our main chapters, and end by discussing further work.

6.1 Results

The work of this thesis consists of three main parts. First a description of RDF-star which adds value to a recent field of research with few publications taking place in Chapter 3. Second an investigation into how to extend VKG systems with RDF-star support, this takes place in Chapter 4. The third part concerns extending the specific VKG system Ontop with RDF-star support and is detailed in Chapter 5.

6.1.1 Theoretical investigation

In order to extend the virtual knowledge graph to an RDF-star graph we have chosen to extend the mapping, specifically the mapping language R2RML. As the mapping forms the basis of creating the virtual graph it seems the natural candidate component. To this end we propose an extension to the standard mapping language R2RML called R2RML-star which allows for the generation of RDF-star triples from database sourced facts.

We detail the impact this has on the query processing of the VKG system including the rewriting and unfolding processes. We further describe other ways in which the system should be extended to fully support RDF- and SPARQL-star, such as new SPARQL-star functions and operators, and extending the query answers.

6.1.2 Extending Ontop

The results presented here are explained in more detail in the various sections of Chapter 5. What has been implemented and is functioning is support for parsing SPARQL-star queries, parsing R2RML-star mappings, and matching the SPARQL-star queries and R2RML-star mappings in a correct way. Unfolding of the query is then supported provided that the queries and mappings map in an exact way and no quoted triples are bound to variables.

Query rewriting of the simple kind from Definition 4.5 is supported. Complex rewriting as in Definition 4.7 remains unsupported as it is unclear if this should be implemented.

The work done for the implemented functionality includes extending Ontop's internal typing system with RDF-star terms and writing test cases, which lays the foundation for further implementing the missing functionality.

6.2 Further Work

We split this section into two parts corresponding to the two main parts of this thesis. The theoretical investigation into applying RDF-star to VKG systems, and the implementation of these ideas to the VKG system Ontop.

6.2.1 Further theoretical work

In this thesis we have not exhausted all the possibilities of applying RDF-star to VKG systems. While we have extended the standard mapping language R2RML in order to add RDF-star support to the mapping of the VKG system, there is another major component left untouched. The ontology could plausibly be extended with support for RDF-star, a work that is left out of this thesis. A theoretical investigation should be made into the applications of RDF-star to the VKG ontology, including if one should extend the standard ontology languages such as RDFS or OWL2QL using RDF-star.

We have not described how to apply query rewriting in the case of complex RDF-star rewriting, as this case is still an open discussion, see Section 4.2.2.

Further, our thesis deals with the extension of VKG system's contextual data capabilities with the help of RDF-star. This necessarily excludes any VKG system not using the RDF language for their knowledge graph. As RDF is a widely used W3C standard we still maintain that this limitation of scope was a correct choice, but we welcome further research into enhancing the contextual data capabilities of VKG systems using other data models.

6.2.2 Further implementation work

Not all of our conclusions of Chapter 4 have been implemented in Ontop. The implementation issue of unfolding mappings that introduce a variable binding to a quoted triple remains unsolved. Also remaining to implement is support for the new SPARQL-star functions, operators, and answer formats. For these additional SPARQL-star features the groundwork has been laid in the new typing system we have developed. All further theoretical investigations suggested in Section 6.2.1 above could also eventually be implemented in Ontop.

Reference List

- [1] Diego Calvanese et al. “Ontop: Answering SPARQL Queries over Relational Databases”. In: *Semantic Web J.* 8.3 (2017), pp. 471–487. DOI: 10.3233/SW-160217.
- [2] Guohui Xiao et al. “Virtual Knowledge Graphs: An Overview of Systems and Use Cases”. In: *Data Intelligence* 1.3 (2019), pp. 201–223. DOI: 10.1162/dint_a_00011.
- [3] Guohui Xiao et al. “The Virtual Knowledge Graph System Ontop”. In: *Proc. of the 19th Int. Semantic Web Conf. (ISWC 2020)*. Vol. 12507. Lecture Notes in Computer Science. Springer, 2020, pp. 259–277. DOI: 10.1007/978-3-030-62466-8_17.
- [4] David Wood, Markus Lanthaler, and Richard Cyganiak. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. URL: <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [5] *SPARQL 1.1 Overview*. W3C Recommendation. W3C, Mar. 2013. URL: <https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>.
- [6] Andy Seaborne and Steven Harris. *SPARQL 1.1 Query Language*. W3C Recommendation. W3C, Mar. 2013. URL: <https://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.
- [7] Dave Beckett and Jeen Broekstra. *SPARQL Query Results XML Format (Second Edition)*. W3C Recommendation. W3C, Mar. 2013. URL: <https://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/>.
- [8] Andy Seaborne. *SPARQL 1.1 Query Results JSON Format*. W3C Recommendation. W3C, Mar. 2013. URL: <https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/>.
- [9] Richard Cyganiak, Seema Sundara, and Souripriya Das. *R2RML: RDB to RDF Mapping Language*. W3C Recommendation. W3C, Sept. 2012. URL: <https://www.w3.org/TR/2012/REC-r2rml-20120927/>.
- [10] Andy Seaborne. *SPARQL 1.1 Query Results CSV and TSV Formats*. W3C Recommendation. W3C, Mar. 2013. URL: <https://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321/>.
- [11] Dan Brickley and Ramanathan Guha. *RDF Schema 1.1*. W3C Recommendation. W3C, Feb. 2014. URL: <https://www.w3.org/TR/2014/REC-rdf-schema-20140225/>.

- [12] Chimezie Ogbuji and Birte Glimm. *SPARQL 1.1 Entailment Regimes*. W3C Recommendation. <https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/>. W3C, Mar. 2013.
- [13] *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation. W3C, Dec. 2012. URL: <https://www.w3.org/TR/2012/REC-owl2-overview-20121211/>.
- [14] Eric Prud'hommeaux and Gavin Carothers. *RDF 1.1 Turtle*. W3C Recommendation. W3C, Feb. 2014. URL: <https://www.w3.org/TR/2014/REC-turtle-20140225/>.
- [15] Olaf Hartig et al. *RDF-star and SPARQL-star*. W3C Draft Community Group Report. W3C Community, July 2021. URL: <https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html>.
- [16] Anastasia Dimou and Miel Van Der Sande. *RDF Mapping Language (RML)*. Unofficial W3C Draft. Oct. 2020. URL: <https://rml.io/specs/rml/>.
- [17] Lisa Ehrlinger and Wolfram Wöb. “Towards a Definition of Knowledge Graphs”. In: Sept. 2016.
- [18] Michael Färber et al. “Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO”. In: *Semantic Web 9* (Mar. 2017), pp. 1–53. DOI: 10.3233/SW-170275.
- [19] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 0124160441.
- [20] *The Description Logic Handbook: Theory, Implementation and Applications*. 2nd ed. Cambridge University Press, 2007. DOI: 10.1017/CB09780511711787.
- [21] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201537710.
- [22] Ontop. *Ontop - Main Features*. 2021. URL: <https://ontop-vkg.org/guide/#main-features> (visited on 05/21/2021).
- [23] Ontop. *Formal characterization of IQs*. 2020. URL: <https://ontop-vkg.org/research/iq-formal.html> (visited on 10/03/2021).
- [24] Ontop. *Intermediate Query*. 2020. URL: <https://ontop-vkg.org/dev/internals/iq.html> (visited on 10/03/2021).
- [25] RDF4J Team. *RDF4J Documentation*. 2021. URL: <https://rdf4j.org/documentation/> (visited on 08/09/2021).
- [26] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and Complexity of SPARQL”. In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009). ISSN: 0362-5915. DOI: 10.1145/1567274.1567278. URL: <https://doi.org/10.1145/1567274.1567278>.

- [27] Olaf Hartig. “Foundations of RDF* and SPARQL* (An Alternative Approach to Statement-Level Metadata in RDF)”. In: *AMW*. 2017. URL: <http://ceur-ws.org/Vol-1912/paper12.pdf> (visited on 07/07/2021).
- [28] Thomas Delva et al. “RML-star: A Declarative Mapping Language for RDF-star Generation”. In: Aug. 2021.
- [29] Guohui Xiao et al. “Ontology-Based Data Access: A Survey”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 5511–5519. DOI: 10.24963/ijcai.2018/777. URL: <https://doi.org/10.24963/ijcai.2018/777>.

Reading List

- [30] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. “Reifying RDF: What Works Well With Wikidata?” In: *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems*. Ed. by Thorsten Liebig and Achille Fokoue. Vol. 1457. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 32–47.
- [31] Claudio Gutierrez et al. “Foundations of Semantic Web databases”. In: *Journal of Computer and System Sciences* 77.3 (2011). Database Theory, pp. 520–541. ISSN: 0022-0000. DOI: <https://doi.org/10.1016/j.jcss.2010.04.009>. URL: <https://www.sciencedirect.com/science/article/pii/S0022000010000516>.
- [32] Graham Klyne and Jeremy Carroll. “Resource Description Framework (RDF): Concepts and Abstract Syntax”. In: *World Wide Web Consortium* 10 (Jan. 2006).

A RDF Graphs

Note that all of these graphs are written in the Turtle syntax. We imagine the following prefixes:

```
@prefix : <http://lukas.thesis.org/films#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://lukas.thesis.org/films#> .
```

RDF Graph 1 (Barebones film ontology)

```
<http://lukas.thesis.org/films> rdf:type owl:Ontology .
```

Object Properties

```
:directedBy rdf:type owl:ObjectProperty ;
             rdfs:domain :Film ;
             rdfs:range :Director .
:stars rdf:type owl:ObjectProperty ;
        rdfs:domain :Film ;
        rdfs:range :Actor .
```

Data properties

```
:releasedIn rdf:type owl:DatatypeProperty ;
             rdfs:domain :Film .
```

Classes

```
:Actor rdf:type owl:Class ;
        rdfs:subClassOf :Person .
:Director rdf:type owl:Class ;
           rdfs:subClassOf :Person .
:Film rdf:type owl:Class .
:Person rdf:type owl:Class .
```

B R2RML Mappings

In addition to the prefixes of appendix A, we add two more:

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.com/ns#>.
@prefix star: < https://w3id.org/obda/r2rmlstar#>.
```

Mapping 1 (Film class with name and release year)

```

<#TriplesMap1>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
    rr:class :Film;
  ];
  rr:predicateObjectMap [
    rr:predicate ex:name;
    rr:objectMap [ rr:column "Name" ];
  ];
  rr:predicateObjectMap [
    rr:predicate :releasedIn;
    rr:objectMap [ rr:column "Year"];
  ].

```

Mapping 2 (R2RML-star example 1)

```

<urn:MAPID-film-source> a rr:TriplesMap;
  rr:logicalTable [ a rr:R2RMLView;
    rr:sqlQuery "SELECT name, year FROM imdbTable1;"
  ];
  rr:subjectMap [ a rr:SubjectMap;
    rr:termType star:RDFStarTermType;
    star:subject [a rr:ObjectMap;
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
    ];
    star:predicate [a rr:PredicateMap;
      rr:constant rdf:type
    ];
    star:object [a rr:ObjectMap;
      rr:constant :Film
    ];
  ];
  rr:predicateObjectMap [
    rr:predicate :source;
    rr:object "IMDB database";
  ].

```

Mapping 3 (R2RML-star example 2)

```

<urn:MAPID-film-mentioned> a rr:TriplesMap;
  rr:logicalTable [ a rr:R2RMLView;
    rr:sqlQuery "SELECT name, year FROM professors;"
  ];

```

```
rr:subjectMap [ a rr:SubjectMap;
  rr:template "http://lukas.thesis.org/films/{Name}{Year}";
  rr:class :Film;
];
rr:predicateObjectMap [
  rr:predicate :isMentionedBy;
  rr:objectMap [ a rr:ObjectMap;
    rr:termType star:RDFStarTermType;
    star:subject [a rr:ObjectMap;
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      rr:termType rr:IRI;
    ];
    star:predicate [a rr:PredicateMap;
      rr:constant :releasedIn;
    ];
    star:object [a rr:ObjectMap;
      rr:column "year";
      rr:termType rr:Literal;
    ];
  ];
].
```

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Database and Ontop Setup

In this tutorial, we run a *DuckDB* database of museums that includes information about the museums themselves, their employees and their exhibits.

Database setup

Procedure to set up the database for the following exercises:

1. Unzip the archive of DuckDB (*duckdb.zip*).
2. That's it! The *data* directory contains the already prepared databases, compatible with *DuckDB version 0.7.1*.
 - If you want to have a first look at the database, you can open it with the DuckDB client application that can be downloaded from the [official site](#).

Ontop SPARQL endpoint setup

For this tutorial, we use a SPARQL endpoint provided by ontop to test our mappings and lenses. Here, we walk through the preparation of the endpoint via *Ontop CLI*, but a docker-image-based endpoint can be created in a similar fashion, as explained in the [endpoint tutorial](#).

1. Download [Ontop CLI](#) and unzip it to a directory, which is denoted as `$ONTOP_CLI_DIR` below.
2. Copy the DuckDB jdbc driver to `$ONTOP_CLI_DIR/jdbc` from the *duckdb.zip* archive.
3. Copy the directory `data` containing the database file `data/tutorial.db` into `$ONTOP_CLI_DIR`.
4. Paste your *mapping* and *lenses* files, as well as the *ontology* and *properties* files into a new directory called `input` inside of `$ONTOP_CLI`.

You will create the *mapping* and *lenses* files throughout the course of this tutorial. Alternatively, the *duckdb.zip* archive contains one directory for each of the lens tutorials, with sample mappings and lenses already pre-made.

- The *ontology* and *properties* files will be the same throughout all tutorials. They are already provided in the *duckdb.zip* archive.

5. Start the Ontop endpoint. On Mac/Linux:

```
$ONTOP_CLI_DIR/ontop endpoint \
  --ontology=input/ontology.ttl \
  --mapping=input/mapping.obda \
  --lenses=input/lenses.json \
  --properties=input/duckdb.properties \
  --cors-allowed-origins=http://yasgui.org # if
needed
```

On Windows:

```
ontop endpoint ^
  --ontology=input/ontology.ttl ^
  --mapping=input/mapping.obda ^
  --lenses=input/lenses.json ^
  --properties=input/duckdb.properties ^
  --cors-allowed-origins=http://yasgui.org
```

6. Open the web interface of the SPARQL endpoint to try some queries: <http://localhost:8080/>

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Using lenses](#)

[Basic Lens →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

Database and Ontop Setup

In this tutorial, we run a *DuckDB* database of museums that includes information about the museums themselves, their employees and their exhibits.

Database setup

Procedure to set up the database for the following exercises:

1. Unzip the archive of DuckDB (*duckdb.zip*).
2. That's it! The *data* directory contains the already prepared databases, compatible with *DuckDB version 0.7.1*.
 - If you want to have a first look at the database, you can open it with the DuckDB client application that can be downloaded from the [official site](#) .

Ontop SPARQL endpoint setup

For this tutorial, we use a SPARQL endpoint provided by ontop to test our mappings and lenses. Here, we walk through the preparation of the endpoint via *Ontop CLI*, but a docker-image-based endpoint can be created in a similar fashion, as explained in the [endpoint tutorial](#).

1. Download *Ontop CLI* and unzip it to a directory, which is denoted as `$ONTOP_CLI_DIR` below.
2. Copy the DuckDB jdbc driver to `$ONTOP_CLI_DIR/jdbc` from the *duckdb.zip* archive.
3. Copy the directory `data` containing the database file `data/tutorial.db` into `$ONTOP_CLI_DIR`.
4. Paste your *mapping* and *lenses* files, as well as the *ontology* and *properties* files into a new directory called `input` inside of `$ONTOP_CLI`.

You will create the *mapping* and *lenses* files throughout the course of this tutorial. Alternatively, the *duckdb.zip* archive contains one directory for each of the lens tutorials, with sample mappings and lenses already pre-made.

- The *ontology* and *properties* files will be the same throughout all tutorials. They are already provided in the *duckdb.zip* archive.

5. Start the Ontop endpoint. On Mac/Linux:

```
sh
$ONTOP_CLI_DIR/ontop endpoint \
  --ontology=input/ontology.ttl \
  --mapping=input/mapping.obda \
  --lenses=input/lenses.json \
  --properties=input/duckdb.properties \
  --cors-allowed-origins=http://yasgui.org # if
needed
```

On Windows:

```
ontop endpoint ^
  --ontology=input/ontology.ttl ^
  --mapping=input/mapping.obda ^
  --lenses=input/lenses.json ^
  --properties=input/duckdb.properties ^
  --cors-allowed-origins=http://yasgui.org
```

6. Open the web interface of the SPARQL endpoint to try some queries: <http://localhost:8080/>

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Using lenses](#)

[Basic Lens →](#)

Tutorial

Presentation

Basics

Database and Ontop Setup

First data source: university 1

Second data source: university 2

Endpoint

Deploying a SPARQL endpoint

Setting up an Ontop SPARQL endpoint with Ontop CLI

Setting up an Ontop SPARQL endpoint with Docker

Interact

Interact with an Ontop SPARQL Endpoint

Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

Second session: Mapping Engineering

Role of primary keys (unique constraints)

Role of foreign keys

Choice of the IRI templates

Bonus: existential reasoning

Materialize

How to materialize data into a graph database

Federation

Federating multiple databases

Ontop with Teiid

In this tutorial, we present how to use Teiid as a federator for Ontop. We provides two options:

- Option 1: Using Docker-compose (recommended)
- Option 2: Manual set up

Option 1: Using Docker-compose

Requirements

- [Docker](#) , version 17.09.0 or higher
- [Docker Compose](#) , version 1.17.0 or higher

Steps

1. Download the code and cd to the directory of the teiid demo

```
$ git clone https://github.com/ontop/ontop-tutorial/
$ cd ontop-tutorial/federation/teiid
```

2. to start the prototype, downloading / building the required images and containers if needed

```
docker-compose up
```

(note: may add option `-d` to run in background, in which case logs are not be displayed to standard output but are still accessible via `docker-compose logs`)

Services When running, the prototype exposes the following services:

- a PostgreSQL server with the sample data, with connection information defined in the `.env` file.
- a Web portal of the SPARQL endpoint backed by ontop at URL <http://localhost:8880/>
- a SPARQL endpoint backed by ontop at URL <http://localhost:8880/sparql> .

3. to stop the prototype, if running

```
docker-compose down
```

4. to stop the prototype, if running, and also clean any image / container / data associating to it (useful for cleaning up)

```
docker-compose down --volumes --remove-orphans
```

(note: the above command does not remove Docker images that may result being unused after stopping and removing this prototype containers; to remove such images, add option `--rm all`)

5. to check the status of the containers forming the prototype

```
docker-compose ps
```

6. to check the logs of specific container(s) or of all containers (if no container name is supplied)

```
docker-compose logs <container name 1> ...  
<container name N>
```

Option 2: Manual set up

In this tutorial, we present step-by-step way of connecting Teiid to Ontop. We show how to integrate *uni1* data stored in a MySQL

DB and *uni2* data stored in a PostgreSQL DB into one Teiid virtual database, and access these two data sources using Ontop in a uniform way.

Install Teiid

Download the latest version 16.0.0 of Teiid from https://teiid.io/teiid_wildfly/downloads/ (choosing Teiid with WildFly/Console, by this way, we do not to install WildFly anymore). Unzip it to a directory, which is denoted as \$TEIID-DIR below.

Configure the environment variable 'JBOSS_HOME' of WildFly:

```
export JBOSS_HOME=$TEIID_DIR
export PATH=$PATH:$JBOSS_HOME/bin
```

Then start 'WildFly' by executing the following commands:

```
~ cd $TEIID_DIR/bin/
~ bin ./standalone.sh
```

And access <http://localhost:8080/> through your browser. If you see the following Welcome page then 'WildFly' has been installed successfully.



After 'WildFly' has been installed, Teiid standalone mode can be started by executing the commands below:

```
~ cd $TEIID_DIR/bin/
~ bin ./standalone.sh -c=standalone-teiid.xml
```

Run 'add-user.sh' (in the directory \$TEIID_DIR/bin/) and follow the introduction to respectively add 'Management User' and 'Application User'. 'Management User' is used to administrate your data sources and VDBs via web Console, and 'Application User' is used to connect Teiid Server.

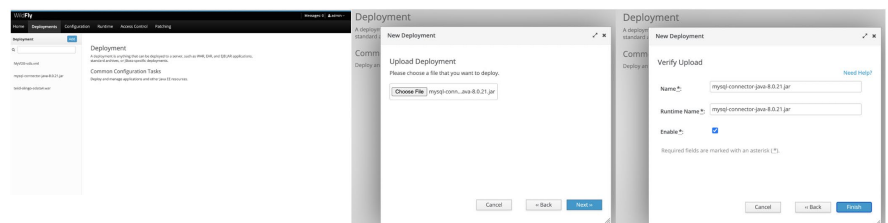
Configure data sources

Before building and deploying VDBs of Teiid, we first need to configure the data sources needed to be integrated. First, start Teiid.

Deploy the drivers of the data sources

Access <http://localhost:8080/> through your browser. Click 'Administration Console' and login with the management user account you have created. Then do the following steps:

- Click 'Deployments'
- Click 'Add'
- Upload the jdbc driver of MySQL, i.e., mysql-connector-java-8.0.21.jar
- Click 'Next'
- Click 'Finish'

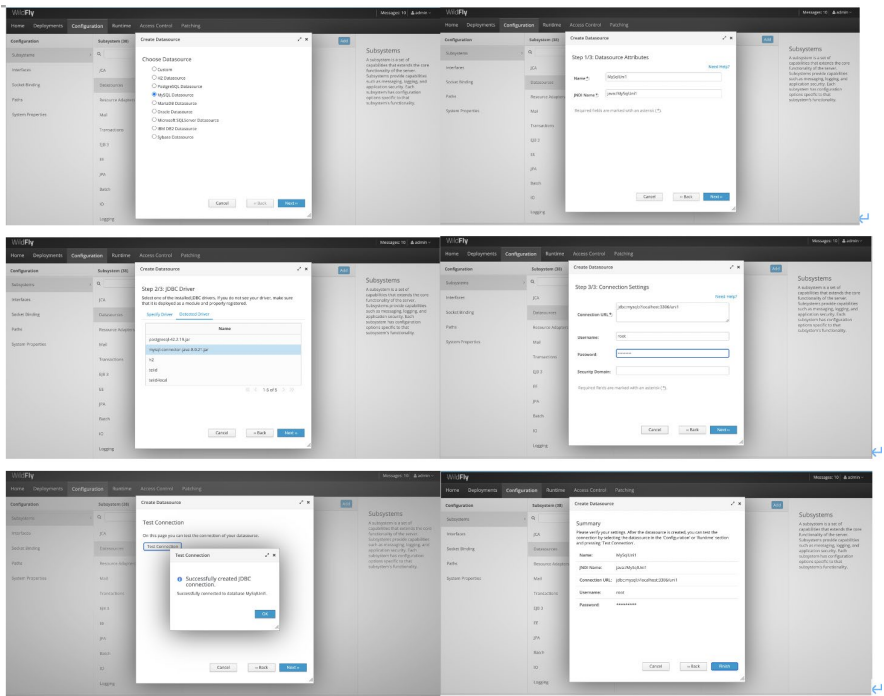


Then deploy the driver of PostgreSQL, i.e., postgresql-42.2.19.jar, in the same way

Configure the data sources

Configure the DB *uni1* of MySQL via the 'Web Console' by the following steps:

- Click 'Configuration'
- Click 'Subsystems'
- Click 'Datasources'
- Click 'Non-XA'
- Click 'Add'
- Choose the type of the 'Datasources', i.e., MySQL
- Click 'Next'
- Input the Name and JNDI Name of the datasource and click 'Next'. Please record the name and JNDI name you inout for the source, since they will be used when build VDBs of Teiid
- Choose the driver of the datasource and click 'Next'
- Input the connection URL, user name and password of the datasource and click 'Next'
- Test the connection of the data source and click 'Next' if the connect is sucessful
- Click 'Finish'



And configure the DB *uni2* of PostgreSQL in the same way.

Build and Deploy VDBs

Create a virtual database (VDB) of Teiid that integartes the MySQL DB *uni1* and PostgreSQL DB *uni2* (the file 'UniversityDBs-

vdb.xml' in 'ontop-
website/.vuepress/public/tutorial/federation/teiid/data/').

```
<vdb name="UniversityDBs" version="1">
  <model visible="true" name="uni1">
    <source name="uni1" translator-
name="mysql5" connection-jndi-
name="java:/MySqlUni1"/>
    <metadata type="DDL"><![CDATA[
      CREATE FOREIGN TABLE student (
        s_id integer not null primary
key,
        first_name varchar(40) not null,
        last_name varchar(40) not null
      )OPTIONS(UPDATABLE 'FALSE');

      CREATE FOREIGN TABLE academic (
        a_id integer not null primary
key,
        first_name varchar(40) not null,
        last_name varchar(40) not null,
        position integer not null
      )OPTIONS(UPDATABLE 'FALSE');

      CREATE FOREIGN TABLE course (
        c_id integer not null primary
key,
        title varchar(100) not null
      )OPTIONS(UPDATABLE 'FALSE');

      CREATE FOREIGN TABLE teaching (
        c_id integer not null,
        a_id integer not null
      )OPTIONS(UPDATABLE 'FALSE');

      CREATE FOREIGN TABLE
course_registration (
        c_id integer not null,
        s_id integer not null
      )OPTIONS(UPDATABLE 'FALSE');
    ]]> </metadata>
  </model>
  <model visible="true" name="uni2">
    <source name="uni2" translator-
name="postgresql" connection-jndi-
name="java:/PostgresUni2"/>
```



```

<metadata type="DDL"><![CDATA[
    CREATE FOREIGN TABLE person (
        pid integer not null primary
key,
        fname varchar(40) not null,
        lname varchar(40) not null,
        status integer not null,
        ssn varchar(40) not null unique
    )OPTIONS(UPDATABLE 'FALSE');

    CREATE FOREIGN TABLE course (
        cid integer not null primary key,
        lecturer integer not null,
        lab_teacher integer not null,
        topic varchar(100) not null
    )OPTIONS(UPDATABLE 'FALSE');

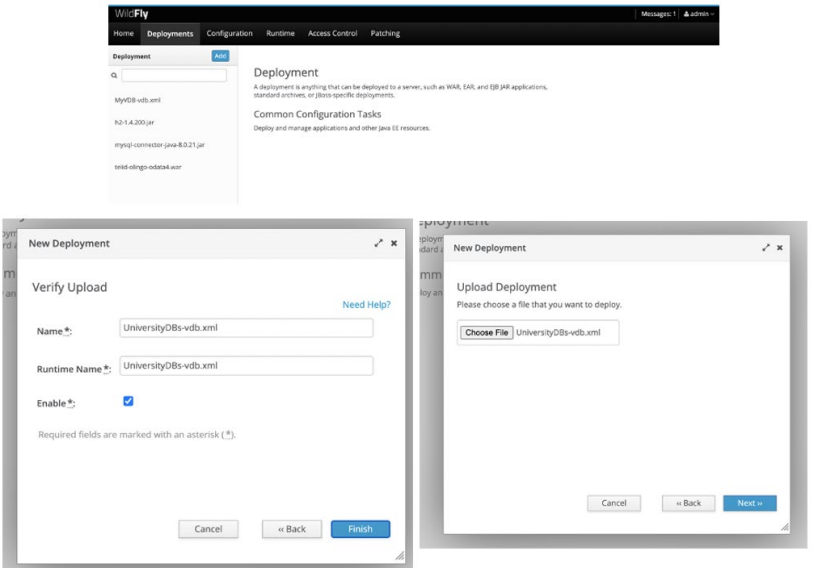
    CREATE FOREIGN TABLE registration (
        pid integer not null,
        cid integer not null
    )OPTIONS(UPDATABLE 'FALSE');
]]> </metadata>
</model>
</vdb>

```

Here, "UniversityDBs" is the name of this VDB. The title of this file should be UniversityDBs-vdb.xml. Note that the values of *connection-jdni-names* must be the names used when configure the corresponding data sources. Here, the foreign tables described by DDL are the schemas of the tables in the DB uni1 and the DB uni2. See http://teiid.github.io/teiid-documents/16.0.x/content/reference/as_virtual-databases.html for more information on how to develop VDBs of Teiid.

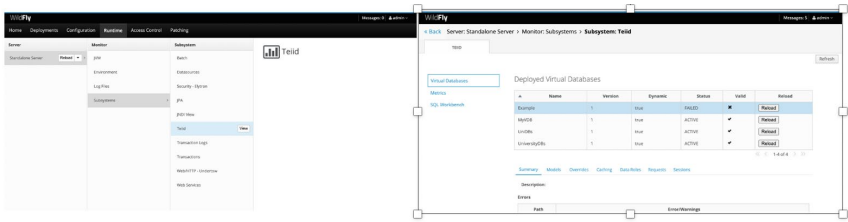
Then deploy the VDB *UniversityDBs-vdb.xml* through the 'Web Console' by the following steps:

- Click 'Deployments'
- Click 'Add'
- Upload the VDB file
- Click 'Next' to verify your VDB
- Click 'Finish'



By the following steps to see whether the VDB has been deployed successfully:

- Click 'Runtime'
- Click 'Standalone Server'
- Click 'Subsystems'
- Choose 'Teiid' and click 'View':



If the 'Statues' of the VDB is 'ACTIVE' then the VDB has been successfully deployed.

Configure Ontop-protege to use the VDB of Teiid

Download the JDBC Driver of Teiid from https://teiid.io/teiid_wildfly/downloads/ , and put it in a directory. Suppose the path is '\$Dir/teiid-16.0.0-jdbc.jar'.

Install the Teiid JDBC driver. Within Protege, clike 'File', 'Reference', 'JDBC Drivers' and 'Add' in turn:

```
. Description: Teiid
. Class name: org.teiid.jdbc.TeiidDriver
```

```
. Driver File (jar): browse to `${Dir}/teiid-16.0.0-jdbc.jar`
```

For the connection, in your Datasource manager, use:

```
. Connection url:
jdbc:teiid:UniversityDBs@mm://localhost:31000
. Database Username: $var1
. Database Password: $var2
. Driver class: org.teiid.jdbc.TeiidDriver
```

Here, 'Var2' are respectively the user name and password of your 'Application User' created before.

Create the ontology and mapping and try some SPARQL queries

You can directly use the files 'university.ttl', 'university.obda' and 'teiid.properties' (in 'ontop-website/.vuepress/public/tutorial/federation/teiid/data/') we have prepared.

Setup a SPARQL endpoint with Command Line Interface (CLI).

Configure Ontop CLI by adding the Teiid JDBC driver:

```
cp ${Dir}/teiid-16.0.0-jdbc.jar
<OntopInstallationPath>/jdbc
```

Run the Ontop CLI:

```
ontop endpoint -t university.ttl -m
university.obda -p teiid.properties --cors-
allowed-origins='*'
```

Access the SPARQL endpoint. Open <http://localhost:8080/>, and run an example SPARQL query "asking for the courses that are taught at both university1 and university as well as the lectures of such courses in university2":

```
PREFIX : <http://example.org/voc#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?x ?z ?y ?e
{?x rdf:type :Course . ?x :title ?y . ?x
:isGivenAt
<http://example.org/voc#uni1/university> .
?z rdf:type :Course . ?z :title ?y . ?z :isGivenAt
<http://example.org/voc#uni2/university> .
?e :givesLecture ?z. }
```

you can obtain the following certain answers which are generated by extracting and merging the data obtained from uni1 and uni2:

```
[x z y e]
[x/<http://example.org/voc#uni1/course/1602>,z/<http://
security",e/<http://example.org/voc#uni2/person/1>]

[x/<http://example.org/voc#uni1/course/1601>,z/<http://
Systems",e/<http://example.org/voc#uni2/person/7>]
```

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Ontop with Dremio](#)

[Using lenses →](#)

Guide

Introduction

Key concepts

Getting started

Command Line Interface

Standards compliance

Advanced

Ontop Mapping Language

Predefined query endpoint (beta)

Query logging

HTTP caching

Configuration keys

Lenses

Data sources ►

Troubleshooting

FAQ

General

Protégé

Meta

Glossary

Release notes

FAQ

Some Frequently Asked Questions

General

Which mapping syntaxes are supported?

We support [R2RML](#) and [our own mapping syntax](#), which can be automatically converted into R2RML using Ontop CLI command

```
ontop mapping to-r2rml
```

Is it open source?

Yes! Ontop is available under the Apache License, Version 2.0.

How can I report a bug or ask other questions?

You can get in touch with us through [our support channels](#) and [contribute to the project](#) by, for instance, filing a [bug report](#).

Does Ontop support SPARQL Update?

No, as for now, Ontop is a read-only system, it does not support [SPARQL Update queries](#) but only SELECT, CONSTRUCT, ASK and DESCRIBE queries. For more details, see [the compliance page](#).

How to set the log level to debug?

For both the Docker image and the CLI, set the environment variable `ONTOP_LOG_LEVEL` to `debug`.

Protégé

No suitable driver found for .. (driver)

You need to download and install the right JDBC driver in Protégé. This is required in order to query using SPARQL and use other features of Ontop. To add a new JDBC Driver:

- in Protégé, go to *Preference -> JDBC Drivers* tab;
- add a new entry;
- type a description, e.g., *PostgreSQL JDBC Driver*;
- type the driver's class string, e.g., *org.postgresql.Driver*;
- type or select the location of the driver's JAR file, e.g., *postgresql-8.4-701.jdbc4.jar*;
- click OK .

See also [Setting up JDBC Drivers](#) and [JDBC Information](#) .

No "JDBC drivers" tab in the preferences

You probably can see the following message in the Protégé console:

```
org.osgi.framework.BundleException: Activator
start error in bundle org.protege.osgi.jdbc.prefs
[20].)
org.protege.osgi.jdbc.RegistryException:
java.lang.ClassNotFoundException: org.h2.Driver
```

Please go to the *Preferences* and click on "*Reset preferences*".

Protégé does not show the Ontop tabs

To see the tabs go to *Windows -> tabs -> Ontop Mappings and Ontop SPARQL*.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

Community

- People
- Organizations
- Support

Contributing

- How to contribute
- Bug report
- Pull request
- Contribution License Agreement (for core developers)
- Documentation

Pull request

Contributions to the project are welcome and encouraged. You can propose some Pull Requests (PR) on Github. By making a PR on the Github platform, you declare to agree with the standard "*inbound = outbound*" licensing practice, as foreseen by the [Github terms](#) . There is no need anymore for regular contributors to sign a CLA.

If you are thinking about implementing a new feature, feel free to engage a discussion with the team through the mailing list or through another medium.

Contribution License Agreement (for core developers)

Core developers should sign a Contribution License Agreement (CLA) that allows FUB and the Ontop team to redistribute their work under the terms of the Apache 2.0 license. If the contributor works for an institution that owns the work being contributed by the contributor, then a corporate CLA must be additionally signed. The CLAs are available for download below, once signed please email a scanned copy of them.

- [Individual Contributor License Agreement \(CLA\)](#)
- [Corporate License Agreement](#)

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

Community

[People](#)

[Organizations](#)

[Support](#)

Contributing

[How to contribute](#)

[Bug report](#)

[Pull request](#)

[Documentation](#)

[Small improvements](#)

[Testing the documentation locally](#)

Documentation

The source code of the documentation is on [Github](#) and is powered by [VuePress](#).

Once a commit is pushed to the Github repository, this public documentation is automatically rebuilt by Travis CI.

Small improvements

Have you found a typo, a broken URL or another small problem? Please propose a fix immediately (before forgetting it), by clicking on the link [Edit this page](#) at the bottom of the page.

Testing the documentation locally

For a more involved work on the documentation, we recommend you to test your changes on your local machine before pushing them to Github.

```
git clone git@github.com:ontop/ontop-website.git
cd ontop-website

# Install the dependencies (e.g. vuepress)
yarn install

# start the local server
yarn docs:dev

# commit your changes and push them back to Github
```

[VuePress](#) requires Node.js >= 8.6.

[Edit this page](#)

[← Pull request](#)

Guide

- Introduction
- Key concepts
- Getting started
- Tutorial
- Using Ontop
- Command Line Interface
- Standards compliance

Advanced

- Ontop Mapping Language
- Predefined query endpoint (beta)
- Query logging
- HTTP caching
- Configuration keys
- Lenses

Data sources ▶

Troubleshooting

- FAQ

Meta

- Glossary
- Release notes

Getting started

Tutorial

If you are new to Ontop and Virtual Knowledge Graphs, we encourage you to have a look [at the official tutorial](#).

Using Ontop

Ontop is distributed under various forms. They can be downloaded on [Github](#) , [Docker Hub](#) , [Sourceforge](#) and in the Protégé plugin repository.

Mapping designer

For editing and testing your mappings, you can use our plugin of the [Protégé ontology editor](#) . You can download the latest stable release directly from Protégé. Alternatively, pre-releases can be found on [Github](#) and [Sourceforge](#) .

Limited support of advanced features

In particular, the Protégé plugin doesn't support lenses. For advanced usage, we recommend using text editors or considering commercial offerings.

Deployment

Once your mappings and your ontology are ready, you can deploy your VKG as a SPARQL endpoint. The Ontop endpoint is available both as a [CLI command](#) (`ontop endpoint`) and as a [Docker image](#) .

You can also use the Ontop endpoint during development as it

embeds [a nice YASGUI client](#) and an optional portal (since 4.0-beta-1) containing pre-defined queries.

Command Line Interface

Want to materialize your VKG, convert your mappings into R2RML, bootstrap your mappings or start a SPARQL endpoint? You can use [the CLI](#) for that. It can be found on [Github](#) and [Sourceforge](#) .

Former solutions

Historically, Ontop has been made available under other means that we don't recommend anymore.

Java API (not recommended)

It remains possible to use Ontop as a Java API through [RDF4J](#) although we recommend the HTTP SPARQL endpoint as a first option (think of a microservice). Why? Because such a Java API would add many dependencies to your project and constraint you to use certain versions of Java.

Ontop-rdf4j is available on [Maven Central](#) .

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Key concepts](#)

[Command Line Interface →](#)

Tutorial

Presentation

Basics

- Database and Ontop Setup
- First data source: university 1
- Second data source: university 2

Endpoint

- Deploying a SPARQL endpoint
- Setting up an Ontop SPARQL endpoint with Ontop CLI
- Setting up an Ontop SPARQL endpoint with Docker

Interact

- Interact with an Ontop SPARQL Endpoint
- Use Jupyter Notebook with an Ontop SPARQL endpoint

Mapping

- Second session: Mapping Engineering
- Role of primary keys (unique constraints)
- Role of foreign keys
- Choice of the IRI templates
- Bonus: existential reasoning

Materialize

- How to materialize data into a graph database

Federation

- Federating multiple databases

External tutorials

- [Destination tutorial](#) : about lodging businesses, municipalities and weather observations, using standard ontologies (schema.org, GeoSPARQL and Semantic Sensor Network).
- [A tutorial using the Bgee database](#) : Code for the paper [Accessing scientific data through knowledge graphs with Ontop](#)

Edit this page

Last Updated: 2/24/2026, 12:10:01 PM

← [SQL Lens](#)

Research

Research team

Publications

Theory

Notation

Formal characterization of IQs

Notation

Leaf node types

Non-leaf node types

Intermediate query optimization

Formal characterization of IQs

WARNING

TODO: CHECK/UPDATE (2 years old and formatted in AsciiDocs)

This section aims at providing a more precise characterization of an *intermediate query* (IQ in what follows), which is the main query representation format used internally in Ontop.

An IQ is a standard (rooted and ordered) tree representation of an algebraic query expression. More exactly, for each node N of a well-formed IQ Q , the maximal subtree of Q rooted in N represents a valid algebraic expression.

In order to give a more precise characterization of this algebra, we will rely on its similarity with Relational Algebra (RA in what follows), more exactly the select/project/join/rename/union RA with named attributes (and no duplicate attribute), extended with additional operators supported by SQL DBMS (and relevant for Ontop's setting). These additional operators are:

- left outer join (which corresponds to the SPARQL OPTIONAL operator), and
- extended projection

NOTE

As of now, difference is not supported.

For readability, some similarities with the SPARQL language will also be (informally) highlighted.

NOTE: This is only one possible way of characterizing the

meaning of an IQ. Other options would be for instance relying on (a customized version of) the SPARQL algebra, or defining from scratch a full-fledged semantic.

link:notation[The first section] introduces some notational conventions, whereas the following ones present the different types of nodes which may appear in an IQ, and how they relate to RA operands and operators. In order to simplify notation, their interpretation is first given in terms of set semantics (which is also the standard interpretation of RA).

The extension to bag semantics has specific implications in an OBDA setting, and is discussed in link:bagSemantics[a dedicated Section].

Notation

Interpretation of algebraic expressions

We assume the following mutually disjoint sets:

- a countably infinite domain D being queried,
- a countably infinite set V of variables names (similar to SPARQL variables, or to a lesser extent to RA attribute names).
- a finite set P of predicates, and
- For each $p \in P$ with arity $n > 0$, a set $\{p_1, \dots, p_n\}$ of base attribute names.

A database instance will be viewed as a total function I over P , which, to each element p of its domain, associates a set $I(p)$ of partial functions from $\{p_1, \dots, p_n\}$ to D .

The *signature* $\text{sig}(M)$ of an algebraic query expression M is the set of its base operands, and it is assumed that for any well-formed M , $\text{sig}(M) \subseteq P$ holds.

Given a database instance I over P , the *interpretation* $\|M\|$ of a well-formed algebraic expression M is built inductively out of all $I(p)$ such that $p \in \text{sig}(M)$, by application of algebraic operators.

To each element of its domain, $\llbracket \cdot \rrbracket$ associates a (possibly empty) set of partial functions from a unique subset $\text{var}(M)$ of V to D . $\text{var}(M)$ will be called the *projected variables* of M , and does not depend on $\llbracket \cdot \rrbracket$, but only on M .

Interpretation of IQ nodes

If Q is a well-formed IQ, then $\text{nodes}(Q)$ will designate its nodes. And by extension, if $N \in \text{nodes}(Q)$, then $\text{sig}(N)$ (resp. $\llbracket N \rrbracket$ and $\text{var}(N)$) will designate the signature (resp. the interpretation and the projected variables) of the algebraic expression represented by (the maximal subtree of Q rooted in) N .

Boolean expressions

A *boolean expression* in what follows designates a quantifier-free first-order formula, with variables in V , constants in D and predicates among the ones listed [link](#): [in the dedicated section].

For instance, $x \neq y$, $(x \geq 2) \vee (x = z)$ and $\text{IsNotNull}(x)$ are valid expressions.

Given a boolean expression φ and a partial function $f : V \mapsto D$, the formula $\varphi[f]$ is obtained by replacing each variable x appearing in φ by $f(x)$ if defined, and by NULL otherwise. Then the valuation $\text{val}(\varphi[f])$ of $\varphi[f]$ corresponds to the one expected in SQL, and operators like "=" or $\geq$ are polymorphic. For instance, for any $d \in D$, $\text{val}(d \leq d) = \text{TRUE}$, $\text{val}(d \neq d) = \text{FALSE}$, $\text{val}(\text{NULL} = d) = \text{UNKNOWN}$, $\text{val}(\text{NULL} \leq d) = \text{UNKNOWN}$, etc.,

These boolean expressions are explicit RA selection conditions. Implicit selection conditions (exclusively equality between variables and/or constant) may also be expressed, for instance by joining two algebraic expressions with shared variables (natural join).

Leaf node types

Data node

Data nodes are prototypical leaf nodes of IQs. Intuitively, a data node is similar to a SPARQL triple pattern, but extended to n -uples.

Equivalently, a data node may represent the application to some $p \in P$ of a (possibly trivial) selection (σ in RA), followed by a variable renaming (ρ in RA).

Internally, a data node N is characterized (using logical notation) with an expression of the form $p(v_1 \sim, \dots, v_n \sim)$, where $p \in P$, and each $v_i \sim$ is either a constant or a variable, i.e. $v_i \sim \in D \cup V$. In addition, we have $\text{var}(N) = \{v_1 \sim, \dots, v_n \sim\} \cap V$.

Constants and repeated variable names express selection in the expected way.

For instance, if N is characterized by $p(x, x)$, then it represents the RA expression $\rho_{[x/p_1, x/p_2]}(\sigma_{[p_1=p_2]} p)$.

Therefore, if $I(p) = \{ \{p_1 \mapsto 1, p_2 \mapsto 2\}, \{p_1 \mapsto 3, p_2 \mapsto 3\} \}$, then $\|N\| = \{ \{p_1 \mapsto 3, p_2 \mapsto 3\} \}$ will hold.

Second order queries

SPARQL triple patterns allow for querying over the name of an RDF property, i.e. the name of a binary predicate. In an IQ, the data node corresponding to such a triple pattern is characterized by a ternary predicate with a meaningless predicate name (currently "triple"), and the property name variable as first argument. For instance, a data node encoding the SPARQL triple pattern $?s ?p ?o$ (where $?p$ stands for RDF property names) will be characterized by the expression $\text{triple}(p, s, o)$.

This mechanism may be extended in future versions to all predicates. In other words, P may be reduced to a set of one predicate per arity ("pair", "triple", "quadruple", etc), with the current $(n-1)$ ary predicate names as first argument.

Intensional VS extensional data nodes

Todo

Empty node

An empty node E can be viewed as a specific kind of data node (whether it should be considered as intensional or extensional is not important). The evaluation of the subquery rooted in E will always be (similar to) the RA n -ary empty relation, with $n \geq 0$.

In other words, for any database instance, $||E|| = \{\}$ holds (although $\text{var}(E)$ may be nonempty).

Alternatively, if $\text{var}(E) = \{v_1, \dots, v_n\}$, assuming some $p \in P$ with arity n , (the subtree whose only node is) E can be viewed as representing the RA expression $\rho_{[v_1/p_1, \dots, v_n/p_n]}(\sigma_{\text{FALSE}}(p))$.

The behavior of $||E||$ as an operand is similar to the behavior of an empty relation in RA. In particular (see [link:nonLeafNodes](#) [the dedicated sections] below for the definitions of the corresponding operators):

- (Left and right) identity element for union: $||E \text{ UNION } M|| = ||M||$, for any M s.t. $\text{var}(M) = \text{var}(E)$
- (Left and right) absorbing element for a natural join: $||E \text{ JOIN } M|| = ||E||$, for any M
- Left absorbing for left outer join, and right identity element if there is no join condition, or if the condition is met (right absorbing otherwise)
- Fixpoint for selection.

True node

True nodes represent another limit case. The evaluation of the subquery rooted in a true node T will always be (similar to) the RA nonempty but 0-ary relation. Or in other words, $\text{var}(T) = \{\}$, and for any database instance, $||T|| = \{f\}$, where f has an empty domain.

The behavior of $\llbracket T \rrbracket$ as an operand is in most cases similar to the behavior of its RA counterpart. In particular (see [link:nonLeafNodes](#)[the dedicated sections] below for the definitions of the corresponding operators):

- (Left and right) identity element for a natural join: $\llbracket T \text{ JOIN } M \rrbracket = \llbracket M \rrbracket$, for any M
- $\llbracket T \text{ UNION } T \rrbracket = \llbracket T \rrbracket$ (see the section about [link:bagSemantics](#)[bag semantics] for the treatment of cardinalities in this specific case).

As expected, T is also the right identity element for the left outer join operator:

- $\llbracket M \text{ LEFT JOIN } T \rrbracket = \llbracket M \rrbracket$, for any M .

In the case where T is the left operand, we have:

- $\llbracket T \text{ LEFT JOIN } M \rrbracket = \llbracket M \rrbracket$, if $\llbracket M \rrbracket \neq \{\}$
- $\llbracket T \text{ LEFT JOIN } M \rrbracket = \llbracket T \rrbracket$ otherwise.

Note that in RA, because the left outer join operator is traditionally defined in terms of join, projection, union and difference, these two queries would be syntactically invalid (implying a difference and a union of two relations with different arities). In IQs on the other hand, the left outer join is viewed as a primitive operator.

Non-leaf node types

Inner join node

An inner join node N represents the n -ary natural join of the evaluations of its children nodes, Explicit joining conditions can also be attached to it, in the form of a boolean expression $\varphi \sim N \sim$, as defined above.

It can be defined almost identically to its SPARQL counterpart. First, for any two partial functions f and g from V to D , let $f \sim g$ hold iff for all $x \in \text{dom}(f) \cap \text{dom}(g)$, we have $f(x) = g(x)$.

Then algebraic operation represented by an inner join node with explicit joining condition φ , denoted with $\text{JOIN}_{\sim\varphi\sim}$, is defined by:

Then $\|N_1 \text{ JOIN}_{\sim\varphi\sim} N_2\| = \{ f \cup g \mid (f,g) \in \|N_1\| \times \|N_2\| \text{ and } f \sim g \text{ holds and } \text{val}(\varphi[f \cup g]) = \text{TRUE} \}$, and $V(N_1 \text{ JOIN}_{\sim\varphi\sim} N_2) = v(N_1) \cup v(N_2)$.

If there is no explicit condition attached to an inner join node, then it represents the operation $N_1 \text{ JOIN}_{\sim 1=1\sim} N_2$, abbreviated as $N_1 \text{ JOIN } N_2$.

Left join node

A left join node represents the RA binary natural left outer join (or SPARQL OPTIONAL) of the evaluations of its children nodes. The left and right children nodes respectively stand for the left and right part of the join.

Explicit joining conditions may also be attached to it.

As for join nodes, this algebraic operation can be defined similarly to its SPARQL counterpart. An inner join node with explicit joining condition φ , represents the algebraic operation $\text{INNER JOIN}_{\sim\varphi\sim}$, defined by:

$\|N_1 \text{ INNER JOIN}_{\sim\varphi\sim} N_2\| = \|N_1 \text{ JOIN } N_2\| \cup \{ f \mid f \text{ is in } \|N_1\| \mid \text{for all } (g) \in \|N_2\|, f \sim g \text{ or } \text{val}(\varphi[f \cup g]) = \text{TRUE} \text{ does not hold} \}$.

As for join nodes, if there is no explicit condition attached to an inner join node, then it represents the operation $N_1 \text{ INNER JOIN}_{\sim 1=1\sim} N_2$, abbreviated as $N_1 \text{ INNER JOIN } N_2$.

Union node

Construction node

A construction represents a RA extended projection, followed by an attribute renaming.

"Extended" projection means here that some additional operations on the values of the projected attributes can be

expressed, prototypically string concatenation to create URIs out of data values (but also for instance arithmetic operations).

Attribute renaming is always the second operation, i.e. it is always performed on the output of the (extended) projection.

optional projection and renaming

Both operations (extended projection and renaming) are in a sense optional, in that the projection with all selected attributes (and no non-trivial additional operation) is the identity operator, and so is renaming with empty substitution. So in practice, a construction node may encode one of these operations.

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Notation](#)

[Intermediate query optimization →](#)

Development

- Overview
- Git workflow
- Build Ontop Bundles
- Make a new release
- Build snapshots
- Debug Protégé
- Debug Jetty
- Support a new database system
- Statistics

Internals

- Intermediate Query (IQ)
- Guice Conventions

Outdated corner

- Known issues
- RDF
- SPARQL
- Mapping
- OWL API
- Databases and SQL
- R2RML mappings
- Ontop Bootstrapper

Known issues

WARNING

THIS PAGE IS OUTDATED.

TODO: check these points and move them somewhere else if still relevant.

RDF

- GRAPH** is not supported yet

SPARQL

- ORDER BY** doesn't support functions as parameters, accepts variables only. Example:

```
SELECT ...
ORDER BY str(?o)

SELECT ...
ORDER BY xsd:integer(?o)

SELECT ...
ORDER BY (?o1 + ?o2)
```

- REGEX** is not supported by MsSQL and DB2, while the other databases handle it differently: see [[Regex SPARQL filter]]
- URI** with a hierarchical directory path of the form directory/directory/.../name are supported if we use them without prefix or with complete prefix. Example we can query `<http://en.wikipedia.org/wiki/BMW_7_Series#section/help>` using it as it is or with a complete prefix as:

```
PREFIX bmwSection:
http://en.wikipedia.org/wiki/BMW_7_Series#section/
SELECT * WHERE {?x ?y bmwSection:help}
```

It is not supported the form:

```
PREFIX bmw:
http://en.wikipedia.org/wiki/BMW_7_Series#
SELECT * WHERE {?x ?y bmw:section/help}
```

- No support for cast functions (e.g. xsd:string(...), xsd:integer(...)).
- **REPLACE** function is supported with some limitations based on the database used. See [[Replace SPARQL filter]].
- Do not use **SELECT Expressions** with more variable declarations, use the correspondent version with BIND.

Example: Instead of:

```
SELECT ?title (?p AS ?fullPrice) (?fullPrice*?
discount) AS ?customerPrice)
{ ?x ns:price ?p .
  ?x dc:title ?title .
  ?x ns:discount ?discount .
}
```

Use:

```
SELECT ?title (?fullPrice*?discount) AS ?
customerPrice)
{
  ?x dc:title ?title .
  ?x ns:discount ?discount .
  BIND (?p AS ?fullPrice)
  ?x ns:price ?fullPrice .
}
```

- Hash functions in oracle are supported only if DBMS_CRYPTO is enabled by the DBA.

- `Hash`, `timezone` and `uuid` functions are supported based on the database see [[OntopSPARQLFunctions|Supported Sparql Functions]] for more information.
- Using Oracle `strBefore` and `strAfter` SPARQL functions return null instead of an empty string when no result has been found.
- `IF`, `COALESCE`, `EXISTS`, `NOT EXISTS`, `IN`, `NOT IN`, `STRDT`, `timezone` and `SHA384` functions are not supported .

Mapping

WARNING

Are Bnodes supported in the mapping?

OWL API

- `Implementation` of most OWLReasoner methods is missing.
- `Symbol @` When we pass a literal to the OWL API, if it finds an "@" it will interpret it as a language tag. The OWL API says:

```
"If the datatype is rdf:PlainLiteral, and the lexical value contains a language tag then the language tag will be parsed out of the lexical value. For example, "abc@en"^^rdf:PlainLiteral would be parsed into a lexical value of 'abc' and a language tag of 'en'. "
```

Thus, if one expects such symbols in an object property the best solutions are:

1. cast the object to string in the mapping
2. replace the @ in the query with @.

Databases and SQL

- **Registered Keywords** - when you have a column name that is also a registered DBMS keyword, you should enclose that column name in quotes. Supported common keywords are: cast, do, extract, first, following, last, materialized, nulls, partition, range, row, rows, siblings, value, xml.
- **ORDER BY** in H2: in presence of an Union, the last version of H2 (1.4.), it orders both subqueries independently of each other giving wrong results. See SPARQL-compliance test offset-1
- **UNOPTIMIZED SQL OPERATORS/FUNCTIONS**: MIN/MAX, CASE, WHEN clause, DATE FUNCTIONS, NESTED SELECTS, MATCHES, RIGHT/FULL/SELF/CARTESIAN JOIN, SUBJOIN, ALL, ANY, UNION, INTERSECT, MINUS and EXCEPT: Unoptimized sql operators will be transformed in view and generate a result.
- **UNSUPPORTED SQL OPERATORS**: EXISTS, UNIQUE, TOP
- **REGEX OPERATOR** is not supported by MySQL and DB2, while the other databases handle it differently

R2RML mappings

WARNING

Are Bnodes supported in the mapping?

- no support for **inverseExpression**
- predicate cannot be uritemplate (column reference or template declaration)
- object cannot be uritemplate (column reference or template declaration) when predicate is rdf:type
- no support for **bnode** in Ontop
- no support for **sqlversion**
- no support for **graphMaps** (hence context graphs)
- bnode naming is arbitrary
- constant type objects are returned as simple literal. Constant-valued term maps are not considered as having a term type, and specifying rr:termType on these term maps has no effect. For example we return "2011-01-04T00:00:00.0" instead of

"2011-01-04T00:00:00.0" ^ ^xsd:dateTime

The current state of the W3C R2RML compliance tests could be seen [\[\[W3C-R2RML-Compliance| here\]\]](#)

Ontop Bootstrapper

WARNING

Are Bnodes supported in the mapping?

- The BNode syntax generated by Ontop bootstrapper cannot be processed by Ontop
- Foreign keys on multiple columns are not supported

[Edit this page](#)

Last Updated: 2/24/2026, 12:10:01 PM

[← Guice Conventions](#)