

OWLIM-SE



Semantic Repository for RDF(S) and OWL

OWLIM-SE (Standard Edition)
Version 4.3

User Guide

Table of Contents

- OWLIM-SE Fact Sheet
- OWLIM-SE Release notes
- OWLIM-SE Introduction
- OWLIM-SE Usage Scenarios
- OWLIM-SE Reasoner
- OWLIM-SE Indexing Specifics
- OWLIM-SE Installation
- OWLIM-SE Configuration
- OWLIM-SE Administrative Tasks
- OWLIM-SE Access Rights and Security
- OWLIM-SE Full-text Search
- OWLIM-SE Geo-spatial Extensions
- OWLIM-SE RDF Rank
- OWLIM-SE Notifications
- OWLIM-SE Query Behaviour
- OWLIM-SE Plug-in API
- OWLIM-SE Experimental Features
- OWLIM-SE LUBM
- OWLIM-SE Performance Tuning
- OWLIM-SE Performance Highlights
- OWLIM-SE References

OWLIM-SE Fact Sheet

OWLIM-SE is the commercial edition of OWLIM – a high-performance semantic repository created by Ontotext. It is implemented in Java and packaged as a Storage and Inference Layer (SAIL) for the Sesame RDF framework. OWLIM-SE is a native RDF rule-entailment and storage engine. The supported semantics can be configured through rule-set definition and selection. Included are rule-sets for OWL-Horst [25], unconstrained RDFS [11] with OWL Lite [14] and the OWL2 profiles RL and QL [30][31]. Custom rule-sets allow tuning for optimal performance and expressivity.

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

OWLIM-SE can manage billions of explicit statements on desktop hardware and can handle tens of billions of statements on commodity server hardware. According to public evaluation data, OWLIM-SE is the most scalable OWL repository currently available.

OWLIM's web site, <http://www.ontotext.com/owlim>, provides extensive information and references regarding support contacts and mailing lists, documentation and latest performance benchmarks.

- [Features](#)
- [Interfaces, Standards, Requirements](#)
- [Licensing](#)
- [Installation and Usage](#)
- [Credits](#)

Features

The key features of the current release of OWLIM-SE can be summarised as follows:

- **The most scalable semantic repository** in the World, both in terms of the volume of RDF data it can store and the speed with which it can load and do inferencing;
- **Pure Java implementation**, ensuring ease of deployment and portability;
- Compatible with **Sesame 2**, which brings interoperability benefits and support for all major RDF syntaxes and query languages;
- Compatible with **Jena** with a built in adapter layer;
- **Customisable reasoning**, in addition to RDFS, OWL-Horst, and OWL 2 RL support;
- **Optimized owl:sameAs** handling, which delivers dramatic improvements in performance and usability when huge volumes of data from multiple sources are integrated.
- **Clustering support** brings resilience, fail-over and scalable parallel query processing;
- **Geo-spatial extensions** for special handling of 2-dimensional spherical data allowing data using the WGS84 RDF vocabulary to be indexed and processed quickly using a variety of special geometrical query constructions and SPARQL extensions functions;
- **Full-text search** support, based on either Lucene or proprietary search techniques;
- **High performance retraction** of statements and their inferences – so inference materialisation speeds up retrieval, but without delete performance degradation;
- Powerful and expressive **consistency/integrity constraint checking** mechanisms;
- **RDF rank**, similar to Google's PageRank, can be calculated for the nodes in an RDF graph and used for ordering **query results by relevance**, visualisation and any other purposes;
- **RDF Priming**, based upon activation spreading, allows efficient data selection and context-aware query answering for handling huge datasets;
- **Notification** mechanism, to allow clients to react to statements in the update stream.

Interfaces, Standards, Requirements

OWLIM-SE is a Java library without user interface. Sesame's Web UI applications can be used.

Interfaces (API, Web Services): OWLIM implements the SAIL interfaces of Sesame, so it can be used locally (embedded) or remotely via the Sesame HTTP server. Additionally, OWLIM-SE can be used with the Jena framework using the built-in adapter.

Platform: Java JRE version 1.5 onwards (both 32-bit and 64-bit versions). If custom rule-sets are used then a Java JDK version 1.6 onwards is required.

Supported standards: OWLIM is bound to the data and query standards supported by Sesame. RDF is the basic data standard; the supported query languages are: SeRQL and SPARQL 1.1.

Syntaxes: The import and export of all major RDF syntaxes (XML, N3, N-Triples, Turtle, TRIG, TRIX) is supported through Sesame.

Semantics: OWLIM supports RDFS, OWL Horst, OWL Max (RDFS with that part of OWL Lite that can be captured in rules), OWL2 QL and OWL2 RL, along with the ability to create customised rule-sets.

Required Libraries: Sesame (<http://www.openrdf.org>) is an open-source RDF database with support for RDF inference and querying.

OWLIM-SE has been tested with Sesame release 2.6.

Licensing

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

Installation and Usage

OWLIM-SE is distributed as a ZIP archive that contains OWLIM-SE's compiled jar's, all required libraries, configuration files,

documentation, sample code and data. OWLIM-SE is designed to be used as a Storage and Inference Layer (SAIL) for the Sesame openRDF framework – no OWLIM specific interfaces are intended for external use. A repackaged [Sesame 2 SDK](#) is included as part of the OWLIM distribution. Use of the rule-compiler (i.e. custom rule-sets) requires the Java Development Kit (JDK) version 1.6 or later, otherwise a Java Runtime Environment (JRE) version 1.5 or later is required.

The easiest way to get to know OWLIM-SE is to explore the Getting Started application that is included in the release package and documented in [the configuration section](#) – it allows one to load data, evaluate queries, and experiment with the configuration parameters, without the need to understand and compile Java code.

The [installation](#) section provides all the information necessary to setup OWLIM-SE with Sesame.

Credits

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

The development of OWLIM is partly supported by [SEKT](#), [TAO](#), [TripCom](#), [LarKC](#), and other [FP6](#) and [FP7](#) European research [projects](#).

OWLIM-SE Release notes

New features and significant bug-fixes/updates for the last few versions are recorded here.

- [Version 4.3](#)
- [Version 4.2](#)
- [Version 4.1](#)
- [Version 4.0](#)
- [Version 3.5](#)
- [Version 3.4](#)
- [Version 3.3](#)

Version 4.3

Further contributions to the Sesame framework from Ontotext and [Fluid Operations](#) mean that [Sesame version 2.6](#) is included with this version of OWLIM. The following new features are available:

- SPARQL 1.1 Federation support that allows queries to pull together data from any number of distributed SPARQL endpoints
- A new SPARQL repository type to wrap SPARQL endpoints
- Improvements to the parser for controlling the level of literal/data-type validation and the handling of errors
- Many other fixes for compliance with the latest revised SPARQL 1.1 working drafts

OWLIM has now has a plug-in API that allows users to build software components that alter the behaviour of OWLIM. This mechanism can be used to add new features or to improve performance in certain scenarios.

OWLIM also includes the following maintenance updates and fixes:

- OWLIM-205 - Validate literal languages and do not allow invalid language tags to enter the repository
- OWLIM-273 - Potential thread leak in QueryModelConverter
- OWLIM-390 - Counting statements using Sesame API gives strange results.
- OWLIM-419 - Make RepositoryConnection.exportStatements obey the time limit
- OWLIM-426 - Unable to permanently remove predefined namespace definitions
- OWLIM-428 - Explicit axioms don't show up as explicit if they have been inferred before by other axioms
- OWLIM-463 - Clear transaction log in replication cluster if it cannot be initialized
- OWLIM-466 - SesameConnectionImpl.getStatements must return quads, not trips (breaks workbench explore)
- OWLIM-470 - Query with Union and optional returns wrong results
- OWLIM-471 - Can not access new repository when FTS switched on (divide by zero or lockfile locked)
- OWLIM-473 - onto:explicit pseudo-graph does not prevent implicit statements as input for query answering
- OWLIM-475 - Repackaged console.sh in openrdf-console.zip has lost its execute attribute
- OWLIM-476 - Neither of the slf4j jars (api or jdk14) are needed in the war files
- OWLIM-483 - Lost solutions to queries with FROM <...> clause
- OWLIM-485 - Repository with many transactions fails to get restored
- OWLIM-488 - Incorrect behaviour of FROM and FROM NAMED in SPARQL queries
- OWLIM-489 - Predicate list indices do not log statistics
- OWLIM-490 - User-supplied Dataset object on query not properly handled
- OWLIM-491 - Query rewriting in MainQuery.convertToOptimizedForm() converts OR to AND in filters when converting the condition to disjunctive normal form
- OWLIM-495 - Blank node contexts ignored by getStatements()
- OWLIM-501 - Lucene and OPTIONAL query bug
- OWLIM-502 - The database restorer deletes the pso and pos files after second unsuccessful restore
- OWLIM-457 - Validate data-type values at load time
- OWLIM-497 - Update getting-started and add timestamps
- OWLIM-356 - Optimized rule set is not compatible with the rule compiler.
- OWLIM-480 - Make use of the com.ontotext.trree.collections for the predicate map in order to reuse the file header and the common interface

Version 4.2

Ontotext have continued to invest in the Sesame project and are pleased to announce the inclusion of [Sesame version 2.5](#) with this version of OWLIM. The benefits include:

- [SPARQL 1.1 Update](#) - this extension of SPARQL provides a [much more powerful method to modify](#) RDF databases without the requirement for developers to use frameworks and APIs.
- SPARQL 1.1 Query conformance has been updated to the [May 2011 working draft](#), i.e. all the remaining behaviour has been implemented along with all the new SPARQL filter functions.
- The SPARQL protocol has also been updated to [January 2010 working draft](#).
- A new binary RDF serialization format. This format has been derived from the existing binary tuple results format. It's main features are reduced parsing overhead and minimal memory requirements.

As well as integration with the new Sesame APIs and modifications for optimising SPARQL Update, there have also been a number of bug fixes in this version of OWLIM-SE:

- OWLIM-396 - A RuntimeException is thrown in clearNamespaces() in SailConnection
- OWLIM-404 - HashEntityPool fails to store/read its entity index table if its size is more than ~500M
- OWLIM-408 - Getting of default namespace doesn't work
- OWLIM-440 - Can not create geo-spatial index when using OWLIM-SE with Tomcat
- OWLIM-443 - Repository fails to start - entity pool error
- OWLIM-445 - disable-sameAs causing query evaluation to lose bindings
- OWLIM-446 - Query.setIncludeInferred() is ignored
- OWLIM-447 - License file can not be specified - default evaluation license is always used.
- OWLIM-449 - Wrong conversion from int to long in com.ontotext.trree.plugin.lucene.LuceneIterator
- OWLIM-452 - Multiple wrong results are returned for a CONSTRUCT query
- OWLIM-454 - EntityStorageVersion3 fails to restore if a long entity has negative size.
- OWLIM-455 - Cannot put any more statements in AVL tree after ~3.1B statements added during 3.5-to-4.0 conversion
- OWLIM-305 - Rationalise OWLIM vocabulary

Version 4.1

This maintenance release includes Sesame 2.4.2, which fixes several important bugs in SPARQL 1.1 Query support:

- [Query performance degradation with assertions enabled](#)
- [No Query Result if Aggregate Function used with OPTIONAL](#)
- [GROUP BY with complex expression not evaluated correctly](#)
- [SUM\(\) with GROUP BY on empty solution results in error](#)
- [IN operator fails on empty argument list](#)
- [ArbitraryLengthPath with lowerbound 0 fails when no zero-length match is found](#)
- [SPARQL parser constructs incorrect query model for some property paths involving alternatives](#)
- [SPARQL parser fails to introduce implicit grouping on some queries](#)
- [Let SUM operator silently ignore non-numeric arguments](#)

Also included are some updates to OWLIM-SE:

- Unexpected binding returned in a Sparql query with union within an optional expression
- FILTER in OPTIONAL patterns returns incorrect results
- Aggregate SPARQL query fails with IndexOutOfBoundsException
- Default and named graphs set in a SPARQL query are ignored by the Jena connector

Version 4.0

- **BigOWLIM** has been renamed to **OWLIM-SE (standard edition)**. This new brand name better reflects the role of this software component at the heart of the OWLIM family. This component is a fully-featured, standalone semantic repository as well as the engine behind the worker nodes in an OWLIM-Enterprise cluster. It's younger sibling, OWLIM-Lite, is the lighter-weight free-for-use version.
- **Easy to deploy WAR files:** The distribution now includes `openrdf-sesame` and `openrdf-workbench` Web applications pre-configured with OWLIM and ready to deploy. This makes installing OWLIM as a server and creating/administrating OWLIM repositories trivially simple. The WAR files can be found in the `sesame_owlim` directory of the distribution ZIP file. See 'easy install' in the [installation section](#).
- **SPARQL 1.1 Query:** Ontotext has invested significant development resources in the [Sesame project](#) in order bring SPARQL 1.1 support to Sesame and OWLIM. This release includes [SPARQL 1.1 Query](#), but without federation support for the moment. [SPARQL 1.1 Update](#) support will be included in the next release. The new features include:
 - Aggregates
 - Subqueries
 - Negation
 - Expressions in the SELECT clause
 - Property Paths
 - Assignment
 - A short form for CONSTRUCT
 - An expanded set of functions and operators
- The SPARQL 1.1 specification has not yet become a W3C recommendation and continues to evolve. The following known issues apply to this release of OWLIM and Sesame:
 - `fn:concat` is not supported. This was added to the working draft in May, just after the Sesame 2.4.0 release was finalised. It will likely be included in the next Sesame/OWLIM release.
 - Federation is not yet supported. This will be implemented in a later version of Sesame and OWLIM later this year.
 - There are some problems with [complex expressions](#) in the SELECT clause. This should be fixed in the next release of Sesame/OWLIM.
 - Empty IN() and NOT IN() clauses will cause an exception - will be fixed in the next release.
 - Using the aggregate function SUM() will cause an exception if there are no bindings over which to do the summation - will be fixed in the next release.
- **Wider entity IDs:** For very large datasets that contain more than 2^{32} unique entities (URIs, blank nodes and literals), OWLIM can be switched in to a new mode that uses 40bit IDs, thus allowing over 1 trillion unique entities.
- **Access to internal entity IDs:** For some applications, especially where RDF URIs are used to index external data, e.g. some legacy system or some other type of storage, then a special predicate and function can be used to find the internal ID used to index an entity or to find an entity based on its ID. This allows for more efficient indexing in external systems, where an integer index can be used instead of a URI string.
- **Internal performance analytics:** It is now possible to monitor the internal behaviour of OWLIM indices using a JMX interface. Statistics can be accessed that show the cache behaviour (number of hits, misses, reads, writes, etc) and these can be helpful when diagnosing any performance problems when loading datasets or when evaluating certain queries. The statistics can give

some indication on how to fine-tune memory allocation between the various caches/indices.

Version 3.5

This release includes many bug fixes, several new features and updates:

- **Remote notifications:** A new mechanism to complement the existing high-performance 'in-process' notification mechanism. This new mechanism allows clients to subscribe for the given statement patterns to remote BigOWLIM repository instances.
- **Schema editing:** Read-only schemas loaded at database initialisation time allow very fast deletion of (instance) statements by using the 'fact-retraction' method that computes the necessary inferred statements to delete. A new mechanism is provided with this release that allows 'read-only' schema statements to be modified when necessary.
- **Configuration spreadsheet tool:** The memory calculator from previous versions has been updated to estimate appropriate BigOWLIM configurations for the specified hardware, dataset characteristics and selected features.
- **Query optimisations:** Several improvements have been made to query optimisation, including the special case when using ORDER BY with LIMIT/OFFSET.
- **Online documentation:** As well as the PDF format user guides included in the OWLIM distribution zip files, [the latest documentation for all editions of OWLIM is now available online](#).
- **Storage files updated automatically:** There are minor differences in storage file formats between versions. Versions of files back to 3.1 are now detected and updated automatically.
- **owl:sameAs optimisation can be disabled:** The `owl:sameAs` optimisation can now be switched off using the `disable-sameAs` configuration parameter. This update might be useful when using the `empty` or `rdfs` rulesets.
- **Lucene-base full-text search enhancements:** Even more fine-grained control over what to include in the indexed RDF molecule. Separate include/exclude lists are now supported for both predicates traversed and entities visited.
- **All OWLIM plug-ins available with Jena interface:** All the BigOWLIM advanced features are now fully supported when using BigOWLIM with the Jena framework. This includes RDF Rank, RDF Search, Node Search, RDF Priming and Geo-spatial extensions.

Version 3.4

This release includes many bug fixes, several new features and updates:

- **Jena adapter:** Applications which use the Jena framework <http://jena.sourceforge.net/> or Jena-compliant RDF stores can seamlessly switch to BigOWLIM to take advantage of efficient loading and high-performance reasoning. At the same time, Jena's ARQ engine allows BigOWLIM to handle the latest SPARQL 1.1 extensions (e.g. aggregates). The adapter is still a beta version and has not been rigorously tested for conformance yet, but can be used with Joseki to make queries and has successfully passed the 100 Million BSBM SPARQL benchmark.
- **Geo-spatial extensions:** Applications can efficiently make queries involving constraints such as "nearby point" and "within region". Special-purpose indices allow such constraints to be evaluated very efficiently on top of large volumes of location-related data – for example, finding airports within 50 miles of London in the GeoNames dataset becomes 500 faster when compared to the same query evaluated without the special indices.
- **Rule engine enhancements:** The rule-engine now supports the ability to use context as part of rule premises and consequences. This allows for more efficient processing of certain RDFS/OWL constructions, particularly those rules using RDF lists. All included rule-sets have been upgraded to make use of this new expressiveness. As a result, there is now just a single rule-set for OWL2-RL, where in the last version there was a 'conformant' and a 'reduced' version. The new rule engine has led to an improvement in LUBM loading performance of around 22%.
- **OWL2-QL:** This OWL2 profile http://www.w3.org/TR/owl2-profiles/#OWL_2_QL is based on DL-Lite_R, a variant of DL-Lite that does not require the unique name assumption. It is designed to be amenable to implementation on relational databases, due to its suitability for re-writing queries to SQL. This release includes a rule-set for this profile in order to expand the range of standard rule-sets and to give users more flexibility when choosing a balance between complexity of inference and scalability.
- **Enhanced Lucene-based full text search:** More flexibility is enabled for using Lucene full text search. Users can create multiple customised indices and can decide to include URIs or literals, select literals by language tags, and use custom analyzers and scorers. Any number of custom indices can be used within the same query.
- **Auto-restore:** A configurable policy parameter can be used to specify how the user wishes the repository to start after an abnormal termination. By default, the database restorer tool will be run automatically to return the database to the state prior to the stop event, i.e. to the state after the last committed transaction.
- **Simplified 'implicit-only' statement retrieval:** When using the Sesame API to return statements, the 'implicit' pseudo-graph is now used. This is simpler and more consistent with query processing than the old method of invoking `RepositoryConnection.getStatements()` twice.
- **Documentation:** The distribution package includes two new guides: *Replication Cluster Quick Start Guide* that has details on installing and configuring a cluster and *Performance Tuning Guide* that brings together all information for optimising loading time, inference and query processing.

Version 3.3

BigOWLIM 3.3 consolidates a number of advanced new features, some of which have been available in previous versions as prototype implementations. The most important differences, compared to previous versions of BigOWLIM are:

- **Replication cluster:** brings resilience, failover and horizontally scalable parallel query processing. A master node component is included that can manage a cluster of worker nodes (standard BigOWLIM instances) to synchronise updates, cater for node failure, dynamically add/remove worker nodes and distributed query requests. Such a setup allows for massive concurrent query performance where query the number of queries processed per second scales almost linearly with the number of worker nodes.
- **OWL2-RL:** Support for this expressive OWL2 profile http://www.w3.org/TR/owl2-profiles/#OWL_2_RL that is amenable for implementation on rule-engines, but without costly data-type reasoning.

- **High performance retraction:** BigOWLIM uses the policy of total materialisation of inferred knowledge. This has the advantage that all inferred statements are computed at load time, thus allowing query processing to proceed extremely quickly. While BigOWLIM has always been very fast at processing incremental statement insertions (which are monotonic by necessity), fast statement retractions are now also possible, despite not utilising any truth maintenance mechanism. When statements are deleted, a combination of forward and backward chaining is used to update the inferred closure. This is slower than statement insertion, but still fast enough to support hundreds of updates per hour even in a repository holding billions of statements.
- **Full text search:** Two different, but complimentary full text search mechanisms are provided, one proprietary and the other based on Lucene. Text search queries can be embedded in SPARQL queries for powerful, hybrid query expressions.
- **Consistency checks:** Are used to ensure the consistency of the repository. The checks use a syntax similar to entailment rules, but are used to signal a consistency violation when the necessary conditions are met. Consistency checks are used in some of the standard rule sets.
- **RDF Rank:** Is a technique that identifies the more important or more popular entities in the repository by examining how many connections (predicates) link each node with any other node. The popularity of entities can then be used to order query results in a similar way to internet search engines, such as Google's PageRank.
- **RDF Priming:** Allows subsets of statements to be selected as the input to query answering. It is based upon the concept of 'spreading activation' as developed in cognitive science. It allows the 'priming' of large datasets with respect to concepts relevant to the context and to the query.
- **Notification mechanism:** A publish/subscribe mechanism for registering and receiving events from a BigOWLIM repository whenever triples matching a certain graph pattern are inserted or removed. This allows clients to react to events in the update stream and avoid polling the repository.
- **Documentation:** improved user documentation with new quick start guide.
- **partialRDFS:** this flag has been deprecated and the optimizations made available with extra rule-set options.
- **Ontology imports:** importing ontologies can be achieved using a URL as well as a local pathname.
- **Better JDK integration:** custom rule-sets require the Java compiler, but it is no longer necessary to have the tools.jar in the classpath.

OWLIM-SE Introduction

This chapter briefly explains the purpose and the scope of this document, its place in the OWLIM user documentation set, and provides some suggestions for efficient use. It starts with an introduction on the differences between the two editions of OWLIM, which facilitates the understanding of the OWLIM product family, the design objectives behind each of the versions and the rationale behind the structuring of the documentation.

- [OWLIM Editions](#)
- [Purpose, Intended Readership and Overview of this Document](#)
- [How to Use This Document](#)
- [Copyrights and Licensing](#)

OWLIM Editions

OWLIM comes in three major editions: OWLIM-Lite, OWLIM-SE (standard edition) and OWLIM-Enterprise.

OWLIM-Lite and OWLIM-SE are identical in terms of usage and integration. Apart from a few differences in configuration parameters, these editions have the same functionality and implement the same Sesame APIs. However, they use different indexing, inference and query evaluation implementations, which results in different performance, memory requirements, and scalability.

OWLIM-Lite is designed for medium data volumes (below 100 million statements) and for prototyping. Its key characteristics are as follows:

- reasoning and query evaluation are performed in main memory;
- it employs a persistence strategy that ensures data preservation and consistency;
- the loading of data, including reasoning, is extremely fast;
- easy configuration.

OWLIM-SE is suitable for handling massive volumes of data and very intensive querying activities. It is designed as an enterprise-grade database management system. This has been made possible through:

- file-based indices, which enable it to scale to billions of statements even on desktop machines;
- special-purpose index and query optimization techniques, ensuring fast query evaluation against very large volumes of data;
- optimized handling of owl:sameAs (identifier equality) to boost efficiency for data integration tasks;
- efficient retraction of explicit statements and their inferences, which allows for efficient delete operations.

OWLIM-Enterprise is a component that can manage and synchronise multiple OWLIM instances in a resilient and scalable cluster configuration.

Further discussion on the differences between OWLIM's editions can be found in the [OWLIM Primer](#).

Purpose, Intended Readership and Overview of this Document

This document is designed for software engineers, system integrators and system administrators who wish to integrate the OWLIM-SE semantic repository in to their applications or who wish to use OWLIM-SE and Sesame as a stand-alone server for storing and processing structured data. It is also useful for system administrators who need to support and maintain a OWLIM-SE repository. The reader is assumed to be familiar with databases, but not an expert in semantic database systems, semantic information retrieval,

Semantic Web, or OWLIM itself. Less experienced readers are recommended to read the OWLIM Primer where the required minimum of Semantic Web and related concepts is covered, as well as OWLIM basics.

This document has multiple uses:

- To provide the information necessary to use OWLIM-SE repositories via the Sesame APIs;
- To give instructions on managing repositories through the Sesame web front-end;
- For familiarisation with OWLIM-SE's reasoning capabilities and rule language;
- To enable the creation of custom installations and configurations of OWLIM-SE;
- To introduce the most important factors that affect the efficiency of OWLIM-SE and to provide some productivity tips.

How to Use This Document

Readers new to OWLIM should read the OWLIM Primer first and then this document in its entirety. This will guarantee that the necessary basic information has been covered in sufficient depth. After some experience using OWLIM, this guide can be used as a reference for carrying out specific tasks.

The following conventions are used in this document:

- Code examples are listed in a `typewriter-like font`
- Other important terms, when first introduced, will be written in *italics*
- Formulas are always in *italics*
- References are given in square brackets, e.g. [3] means "Refer to publication #3 in the References section"
- OWLIM-Lite, OWLIM-SE and OWLIM-Enterprise are referred to collectively as "OWLIM" in those cases where it is not necessary to distinguish between the editions, i.e. the specifications, features, and/or settings described are common for all editions

Copyrights and Licensing

This document and the products discussed in it are copyrighted and subject to licensing as follows:

- **OWLIM-SE**, © Copyright Ontotext AD. 135 Tsarigradsko Shosse, Sofia 1784, Bulgaria, <http://www.ontotext.com>.
- **Sesame**, © Copyright Aduna b.v. Stadsring 181, 3817 BA Amersfoort, The Netherlands Sesame is an open-source library, available under the GNU Lesser GPL (<http://www.gnu.org/copyleft/lesser.html>)
 - All other trademarks mentioned in this document, belong to their respective owners.

Full licensing information is available at <http://www.ontotext.com/owlim/>, as well as in the licence files located in the main folder of the distribution package.

OWLIM-SE Usage Scenarios

- [Access Methods](#)
- [Sesame Application Programming Interface \(API\)](#)
 - [Using the Sesame API to access a local OWLIM repository](#)
 - [Using the Sesame API to access a remote OWLIM repository](#)
- [Managing repositories with the Sesame Workbench](#)
- [Using OWLIM-SE with Jena](#)

Access Methods

OWLIM version is packaged as a Storage and Inference Layer (SAIL) for [Sesame version 2.x](#) and makes extensive use of the features and infrastructure of Sesame, especially the RDF model, RDF parsers and query engines. Inference is performed at load time, where the explicit and inferred statements are stored in highly-optimized data structures. The inferred closure is updated through inference at the end of each transaction that modifies the repository.

OWLIM implements the Sesame SAIL interface so that it can be integrated with the rest of the Sesame framework, e.g. the query engines and the web user interface tools. A typical user application uses OWLIM directly through the Sesame SAIL API. When an OWLIM repository is exposed using the Sesame HTTP Server, users can manage the repository through the Sesame Workbench Web application, or with other tools that can integrate with Sesame, e.g. ontology editors like TopBraid Composer.

However, OWLIM-SE can also be used with the [Jena](#) framework, which is achieved with a customised Jena/Sesame/OWLIM-SE adapter component.

Sesame comprises a large collection of libraries, utilities and APIs, but the important components for this section are:

- the Sesame classes and interfaces (API) that provide uniform access to SAIL components from multiple vendors/publishers;
- the Sesame server and workbench Web applications (Java Enterprise Edition servlet components, referred to on the diagram below as "Sesame Web UI").

This section will describe using the Sesame API to create and access OWLIM-SE repositories, both on the local file-system and remotely via the Sesame HTTP server, plus a brief introduction will be given to the Sesame workbench Web application that provides many repository management functions through a convenient user interface. This is followed by information on using OWLIM-SE with Jena and some related components, i.e. [ARQ](#) and [Joseki](#).

Sesame Application Programming Interface (API)

Programmatically, OWLIM is used via the Sesame Java framework of classes and interfaces. Documentation for these interfaces (including Javadoc) can be found at <http://www.openrdf.org>. Code snippets in the following sections are taken from (or are variations of) the GettingStarted example program that comes with the OWLIM-SE distribution.

Using the Sesame API to access a local OWLIM repository

With Sesame 2, repository configurations are represented as RDF graphs. A particular repository configuration is described as a resource, possibly a blank node, of type: <http://www.openrdf.org/config/repository#Repository>. This resource has an `id`, a label and an implementation, which in turn has a type, SAIL type, etc. The example repository configuration from the getting-started example program looks like this:

```

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

[] a rep:Repository ;
  rep:repositoryID "owlim" ;
  rdfs:label "OWLIM Getting Started" ;
  rep:repositoryImpl [
    rep:repositoryType "openrdf:SailRepository" ;
    sr:sailImpl [
      sail:sailType "owlim:Sail" ;
      owl:ruleset "owl-horst-optimized" ;
      owl:storage-folder "storage" ;
      owl:base-URL "http://example.org/owlim#" ;
      owl:repository-type "file-repository" ;
      owl:imports "./ontology/owl.rdfs" ;
      owl:defaultNS "http://example.org/owlim#" .
    ]
  ] .

```

The Java code to use the configuration to instantiate a repository and get a connection to it is as follows:

Example Java code to create a repository

```

Graph graph = ...;
Resource repositoryNode = ...;

// Create a manager for local repositories
RepositoryManager repositoryManager = new LocalRepositoryManager(new File("."));
repositoryManager.initialize();

// Create a configuration object from the configuration graph
// and add it to the repositoryManager
RepositoryConfig repositoryConfig = RepositoryConfig.create(graph, repositoryNode);
repositoryManager.addRepositoryConfig(repositoryConfig);

// Get the repository to use
Repository repository = repositoryManager.getRepository("owlim");

// Open a connection to this repository
RepositoryConnection repositoryConnection = repository.getConnection();

// ... use the repository

// Shutdown connection, repository and manager
repositoryConnection.close();
repository.shutdown();
repositoryManager.shutdown();

```

Note that the code to parse this file and find the 'root' node for the configuration can be found in the getting-started program. The procedure is as follows: instantiate a local repository manager with the data directory to use for the repository storage files (repositories will store their data in their own sub-directory from here), add a repository configuration for the desired repository type to the manager, 'get' the repository and open a connection to it. From then on, most activities will use the connection object to interact with the repository, e.g. executing queries, adding statements, committing transactions, counting statements, etc. See the getting-started application for examples.

Using the Sesame API to access a remote OWLIM repository

The Sesame Server is a Web application that allows interaction with repositories using the HTTP protocol. It runs in a JEE compliant servlet container, e.g. Tomcat, and allows client applications to interact with repositories located on remote machines. All that is required to connect to and use a remote repository instead of a local one is to replace the local repository manager for a remote one. The URL of the Sesame Server must be provided, but no repository configuration is needed if the repository already exists on the server. The following lines can be added to the getting-started example program, although a correct URL must be specified:

Example to create a connection to a remote repository

```
RepositoryManager repositoryManager =
    new RemoteRepositoryManager( "http://192.168.1.25:8080/openrdf-sesame" );
repositoryManager.initialize();
```

The rest of the example program should work as expected, although the following library files must be added to the class-path:

- commons-httpclient-3.1.jar
- commons-logging-1.1.1.jar
- commons-codec-1.3.jar

Managing repositories with the Sesame Workbench

The [installation section](#) explains how to set up an OWLIM-SE repository that is exposed via the Sesame HTTP Server. In summary, the Sesame Server and Workbench applications are deployed to a Tomcat instance and the necessary library files (OWLIM-SE, Lucene, etc) are copied to the correct locations. After this, the Sesame console application is used with the `owlim.ttl` repository template file to connect to the Sesame server and create a repository instance.

When the Sesame server is running, it will show a welcome message at the following URL:

```
http://<hostname|ip_address>:8080/openrdf-sesame
```

The server has a simple user interface that shows status, logging and configuration information.

The workbench application, however, provides most repository management functions and is available from the following URL:

```
http://<hostname|ip_address>:8080/openrdf-workbench
```

The workbench lists repositories and their namespaces, allows for the addition and deletion of statements, and provides a query interface for SPARQL and SerQL query languages. However, the list of repository types is currently hard-coded in the workbench, so an OWLIM repository must still be created using the Sesame console application, as described in the [installation section](#).

Using OWLIM-SE with Jena

Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS, OWL and SPARQL and includes a rule-based inference engine. Access to OWLIM-SE via the Jena framework is achieved with a special adapter, which is essentially an implementation of the Jena [ARQ](#) interface that provides access to individual triples managed by a OWLIM-SE repository through the Sesame API interfaces.

The Jena adapter for OWLIM-SE is not a general purpose Sesame adapter and cannot be used to access any Sesame compatible repository, because it utilises an internal OWLIM-SE API to provide more efficient methods for processing RDF data and evaluating queries.

The adapter comes with its own implementation of a Jena 'assembler' factory to make it easier to instantiate and use with those related parts of the Jena framework, although one can instantiate an adapter directly by providing an instance of a Sesame `SailRepository` (an OWLIM-SE `OWLIMRepository` implementation). Query evaluation is controlled by the [ARQ](#) engine, but specific parts of a Query (mostly batches of statement patterns) are evaluated natively through a modified `StageGenerator` plugged into the Jena runtime framework for efficiency. This also avoids unnecessary cross-API data transformations during query evaluation.

The Jena adapter can be used as follows, where a 'repositoryConnection' is obtained as in the example in the Sesame section above:

```
import com.ontotext.jena.SesameDataset;

// Create the DatasetGraph instance
SesameDataset dataset = new SesameDataset(repositoryConnection);
```

From now on the `SesameDataset` object can be used through the Jena API as regular `Dataset`, e.g. to add some data to it one could do something like the following:

```
Model model = ModelFactory.createModelForGraph(dataset.getDefaultGraph());
Resource r1 = model.createResource("http://example.org/book#1");
Resource r2 = model.createResource("http://example.org/book#2");
r1.addProperty(DC.title, "SPARQL - the book");
    .addProperty(DC.description, "A book about SPARQL");
r2.addProperty(DC.title, "Advanced techniques for SPARQL");
```

The performance of OWLIM-SE when used through Jena is quite similar to using it through the Sesame APIs. For most of the scenarios

and tasks OWLIM-SE can deliver considerable performance improvements when used as a replacement for Jena's own native RDF backend TDB. More information is provided on the [OWLIM through Jena Performance page](#).

OWLIM-SE Reasoner

- Rule-Based Inference
- OWLIM-SE's Logical Formalism
 - Rule Format and Semantics
 - The Rule Language
 - Prefices
 - Axioms
 - Rules
 - Entailment rules
 - Consistency checks
 - Materialization
 - Retraction of assertions
 - Schema update transactions
- Predefined Rule-Sets
 - OWL2 QL non-conformance
- Custom Rule-Sets
- Performance Optimizations in RDFS and OWL Support
- owl:sameAs Optimization

Rule-Based Inference

There are two principle strategies for rule-based inference called 'forward-chaining' and 'backward-chaining'. They can be briefly explained as follows:

- **Forward-chaining:** involves applying the inference rules to the known facts (explicit statements) to generate new facts. The rules can then be re-applied to the combination of original facts and inferred facts to produce yet more new facts. The process is iterative and continues until no new facts can be generated. This kind of reasoning can have diverse objectives, for instance: to compute the inferred closure, to answer a particular query, to infer a particular sort of knowledge (e.g. the class taxonomy), etc. The advantage of this approach is that when all inferences have been computed query answering can proceed extremely quickly. The disadvantages come from greater initialisation costs (inference computed at load time) and space/memory usage (especially when the number of inferred facts is very large);
- **Backward-chaining:** involves starting with a fact to be proved or a query to be answered. Typically, the reasoner examines the knowledge base to see if the fact to be proved is present and if not it examines the rule set to see which rules could be used to prove it. For the latter case, a check is made to see what other 'supporting' facts would need to be present to 'fire' these rules. The reasoner searches for proofs of each of these 'supporting' facts in the same way and iteratively maps out a search tree. The process terminates when either all of the leaves of the tree have proofs or no new candidate solutions can be found. Query processing is similar, but only stops when all search paths have been explored. The purpose in query answering is to find not just one, but all possible substitutions in the query expression. The advantages of this approach are that there are no inferencing costs at start-up and minimal space requirements. The disadvantage is that inference must be done each and every time a query is answered and for complex search graphs this can be computationally expensive and slow.

Both of these strategies have their advantages and disadvantages, which have been well studied in the history of KR and expert systems. Attempts to overcome the weak points have led to the development of various hybrid strategies (involving partial forward- and backward-chaining) which have proven efficient in many contexts

Reasoning and materialization are discussed in some detail in the [OWLIM Primer](#) .

OWLIM-SE's Logical Formalism

RDFS inference is achieved via a set of axiomatic triples and entailment rules. These rules allow the full set of valid inferences using RDFS semantics to be determined. Herman ter Horst in [25] defines RDFS extensions for more general rule support and a fragment of OWL, which is more expressive than DLP and fully compatible with RDFS. First, he defines R-entailment, which extends RDFS-entailment in the following ways:

- it can operate on the basis of any set of rules R (i.e. allows for extension or replacement of the standard set, defining the semantics of RDFS);
- it operates over so-called generalized RDF graphs, where blank nodes can appear as predicates (a possibility disallowed in RDF);
- rules without premises are used to declare axiomatic statements;
- rules without consequences are used to detect inconsistency (integrity constraints).

OWLIM uses a notation almost identical to R-Entailment defined by Horst. One major difference is that two forms of consistency checking rules are permitted. The first form is the same as defined by R-entailment, i.e. a rule without a consequence indicates inconsistency when its premises are satisfied. The second form has consequences that identify statements that must exist when the premises are true.

OWLIM-SE performs reasoning based on forward-chaining of entailment rules defined using RDF triple patterns with variables.

OWLIM-SE's reasoning strategy is 'total materialisation', which is introduced in the OWLIM Primer in the Reasoning Strategies topic

Rule Format and Semantics

The rule format and the semantics enforced is analogous to R-entailment (see the Rule-Based Inference topic on page and [25]) with the

following differences:

- Free variables in the head (without binding in the body) are treated as blank nodes. This feature must be used with extreme care, because custom rule-sets can easily be created that recursively infer an infinite number of statements making the semantics intractable;
- Variable inequality constraints can be specified in addition to the triple patterns (they can be placed after any premise or consequence). This leads to lower complexity compared to R-entailment;
- the **[cut]** operator can be associated with rule premises. This is an optimisation that tells the rule compiler not to generate a variant of the rule with the identified rule premise as the first triple pattern;
- Context can be used for both rule premises and rule consequences allowing more expressive constructions that utilise 'intermediate' statements contained within the given context URI;
- Consistency checking rules have two forms: With consequences (used to check that certain inferences have occurred) and without consequences (which indicate inconsistency when the premises are satisfied);
- Axiomatic triples can be provided as a set of statements, although these are not modelled as rules with empty bodies.

OWLIM-SE can be configured via "rule-sets" – sets of axiomatic triples, consistency checks and entailment rules - that determine the applied semantics. The implementation of OWLIM-SE relies on a compile stage, during which the rules are compiled into Java source code that is then further compiled using the Java compiler and merged together with the inference engine.

The Rule Language

A rule-set file has three sections named **Prefices**, **Axioms**, and **Rules**. All sections are mandatory and must appear sequentially in this order.

Prefices

This section defines abbreviations for the namespaces used in the rest of the file. The syntax is:

```
shortname : URI
```

A typical prefixes section might look like this:

```
Prefices
{
  rdf : http://www.w3.org/1999/02/22-rdf-syntax-ns#
  rdfs : http://www.w3.org/2000/01/rdf-schema#
  owl : http://www.w3.org/2002/07/owl#
  xsd : http://www.w3.org/2001/XMLSchema#
}
```

Axioms

This section is used to assert 'axiomatic triples', which are usually used to describe the meta-level primitives used to define the schema, such as `rdf:type`, `rdfs:Class`, etc. This section contains a list of the (variable free) triples, one per line. For example, the RDF axiomatic triples are defined thus:

```
Axioms
{
  // RDF axiomatic triples
  <rdf:type>      <rdf:type> <rdf:Property>
  <rdf:subject>   <rdf:type> <rdf:Property>
  <rdf:predicate> <rdf:type> <rdf:Property>
  <rdf:object>    <rdf:type> <rdf:Property>
  <rdf:first>     <rdf:type> <rdf:Property>
  <rdf:rest>      <rdf:type> <rdf:Property>
  <rdf:value>     <rdf:type> <rdf:Property>
  <rdf:nil>       <rdf:type> <rdf:List>
}
```

Rules

This section is used to define entailment rules and consistency checks, which share a similar format. Each definition consists of premises and corollaries that are RDF statements defined with subject, predicate, object and optional context components. The subject, predicate and object can each be a variable, blank node, literal, full URI or the short name for a URI. If given, the context must be a full URI or a short name for a URI.

If the context is provided, then the statements produced as rule consequences are not 'visible' during normal query answering. Instead, they can only be used as input to this or other rules and then only when the rule premise explicitly uses the given context. See the example below.

Furthermore, inequality constraints can be used to state that the values of the variables in a statement must not be equal to some specific

full URI (or its short name) or to the value of another variable within the same rule. The behaviour of an inequality constraint depends whether it is placed in the body or head of a rule. If it is placed in the body of a rule, then the whole rule will not 'fire' if the constraint fails, i.e. the constraint can be next to any statement pattern in the body of a rule with the same behaviour (the constraint does not have to be placed next to the variables it references). If the constraint is in the head, then its location is significant, because a constraint that does not hold will prevent only the statement it is adjacent to from being inferred.

Entailment rules

The syntax of a rule definition is as follows:

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

Where each premise and consequence is on a separate line. The following example helps to illustrate the possibilities:

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

  Id: rdfs2
    x a y [Constraint a != <rdf:type>]
    a <rdfs:domain> z [Constraint z != <rdfs:Resource>]
    -----
    x <rdf:type> z

  Id: owl_FunctProp
    p <rdf:type> <owl:FunctionalProperty>
    x p y [Constraint y != z, p != <rdf:type>]
    x p z [Constraint z != y] [Cut]
    -----
    y <owl:sameAs> z
}
```

The symbols **p**, **x**, **y**, **z** and **a** are variables. The second rule contains two constraints that reduce the number of bindings for each premise, i.e. they 'filter out' those statements where the constraint does not hold.

In a forward chaining inference step, a rule is interpreted as meaning that for all possible ways of satisfying the premises, the bindings for the variables are used to populate the consequences of the rule. This generates new statements that will manifest themselves in the repository, for example, by being returned as query results.

The last rule contains an example of using the **[Cut]** operator, which is an optimisation hint for the rule compiler. When rules are compiled, a different variant of the rule is created for each premise, so that each premise occurs as the first triple pattern in one of the variants. This is done so that incoming statements can be efficiently matched to appropriate inferences rules. However, when a rule contains two or more premises that match identical triples patterns, but using different variable names, then the extra variant(s) are redundant and better efficiency can be achieved by simply not creating the extra rule variant(s). In the above example, rule `owl_FunctProp` would by default be compiled in to three variants:

```

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

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

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

```

As can be seen, the last two variants are identical apart from the rotation of variables **y** and **z**, so one of these variants is not needed. The use of the **[Cut]** operator above tells the rule compiler to eliminate this last variant, i.e. the one beginning with the premise **x p z**. The use of context in rule bodies and rule heads is also best explained by example. The following three rules implement the OWL2-RL property chain rule (prp-spo2) [31] and are inspired by the Rule Interchange Format (RIF) implementation [27]:

```

Id: prp-spo2_1
  p <owl:propertyChainAxiom> pc
  start pc last                [Context <onto:_checkChain>]
  -----
  start p last

Id: prp-spo2_2
  pc <rdf:first> p
  pc <rdf:rest> t               [Constraint t != <rdf:nil>]
  start p next
  next t last                  [Context <onto:_checkChain>]
  -----
  start pc last                [Context <onto:_checkChain>]

Id: prp-spo2_3
  pc <rdf:first> p
  pc <rdf:rest> <rdf:nil>
  start p last
  -----
  start pc last                [Context <onto:_checkChain>]

```

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

Consistency checks

Consistency checks are used to ensure that the data model is in a consistent state and are applied whenever an update transaction is committed. The syntax is similar to that of rules, except that Consistency replaces the `Id` tag that introduces normal rules and furthermore consistency checks do not need to have any consequences. Consistency checks that have no consequences will indicate an inconsistency whenever their premises can be satisfied, e.g.

```

Consistency: something_can_not_be_nothing
  x rdf:type owl:Nothing
  -----

Consistency: both_sameAs_and_differentFrom_is_forbidden
  x owl:sameAs y
  x owl:differentFrom y
  -----

```

These inconsistency checks will output an error message to **standard output** whenever their premises are satisfied. No other action will be taken (no exception is thrown and the behaviour of the repository is not changed). The second consistency check describes a contradiction, which are typically expressed by inconsistency checks without consequences. Consistency checks can also have consequences. In this case, they behave very much like entailment rules, except that instead of inferring new statements, they are used to confirm the existence of explicit or implicit statements, e.g. the following consistency checks could be used to ensure the correct domain and range of a predicate:

```
Consistency:
  x p y
  p rdfs:domain c
  -----
  x rdf:type c

Consistency: range
  x p y
  p rdfs:range c
  -----
  y rdf:type c
```

Consistency checks can have multiple consequences, but the semantics will remain the same – when all the premises are satisfied and one of the consequences is not found in the repository, then the data is inconsistent and an error message is written to **standard output**. The error message will include the statements that caused the inconsistency. The mechanism of inconsistency checking is switched off by default. It can be switched on by using the boolean parameter **check-for-inconsistencies**, see section 8.5.

Materialization

An OWLIM repository will use the configured rule-set to compute all inferred statements at load time. To some extent, this process increases processing cost and time taken to load a repository with a large amount of data. However, it has the desirable advantage that subsequent query evaluation can proceed extremely quickly.

Apart from a number of optimisations, the approach taken by OWLIM is one of 'total materialization', where the inference rules are applied repeatedly to the asserted (explicit) statements until no further inferred (implicit) statements are produced.

Retraction of assertions

OWLIM stores explicit and implicit statements, i.e. those statements inferred (materialized) from the explicit statements. It follows therefore, that when explicit statements are removed from the repository, any implicit statements that rely on the removed statement must also be removed.

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

Removal of explicit statements is now achieved in a more efficient manner, by invalidating only those inferred statements that can no longer be derived in any way.

One approach is to maintain track information for every statement, typically the list of statements that can be inferred from this statement. The list is built up during inference as the rules are applied and the statements inferred by the rules are added to the lists of all statements that triggered the inferences. The drawback of this technique is that track information inflates more rapidly than the inferred closure - in the case of large datasets up to 90% of the storage is required just to store the track information.

Another approach is to perform backward chaining. Backward chaining does not require track information, since it essentially re-computes the tracks as required. Instead, a flag for each statement is used in order that the algorithm can detect when a statement has been previously visited and so avoid an infinite recursion. The algorithm used in OWLIM-SE works as following:

1. Apply a 'visited' flag to all statements which is 'false' by default;
2. Store the statements to be deleted in the list L;
3. For each statement in L that is not visited yet mark it as visited and apply the forward chaining rules. Statements marked as visited become invisible, which is why the statement must be first marked and then used for forward chaining;
4. If there are no more unvisited statements in L then END;
5. Store all inferred statements in the list L1;
6. For each element in L1 check the following:

If the statement is purely implicit statement (a statement can be both explicit and implicit and if so then it is not considered purely implicit) then mark it as deleted (prevent it from being returned by the iterators) and check whether it is supported by other statements. The `isSupported()` method uses queries which contain the premises of the rules and the variables of the rules are preliminarily bound using the statement in question. That is to say the `isSupported()` method starts from the projection of the query and then checks whether the query will return results (at least one), i.e. this method performs backward chaining.

If a result is returned by any query (every rule is represented by a query) in `isSupported()` then this statement can be still derived from other statements in the repository, so it must not be deleted (its status is returned to 'inferred').

If all queries return no results then this statement can no longer be derived from any other statements, so its status remains 'deleted' and the number of statements counter is updated;

1. L := L1 and GOTO 3.

Special care is taken when retracting `owl:sameAs` statements, so that the algorithm still works correctly when modifying equivalence classes. By default, OWLIM-SE uses the approach described above to dramatically improve performance when statements are deleted from the repository. This behaviour can be switched on and off with the repository parameter `enableSmoothDelete`

described in the [configuration section](#).



One consequence of this algorithm, is that deletion can still have poor performance when deleting schema statements, due to the (probably) large number of implicit statements inferred from them.



The forward chaining part of the algorithm terminates as soon as it detects that a statement is read only, because if it cannot be deleted there is no need to look for statements derived from it. For this reason, performance can be greatly improved when all schema statements are made read-only by importing ontologies (and OWL/RDFS vocabularies) using the `imports repository` parameter.

Schema update transactions

In situations when fast statement retraction is required, but it is also necessary to update schemas, a special statement pattern can be used. By including a statement with the following form in the update:

```
?subject <http://www.ontotext.com/owlim/system#schemaTransaction> ?object
```

where `?subject` and `?object` can be anything, OWLIM will use the smooth-delete algorithm, but will also traverse read-only statements and allow them to be deleted/inserted. Such transactions are likely to be much more computationally expensive to achieve, but are intended for the occasional, offline update to otherwise read-only schemas. The advantage is that fast-delete can still be used, but a repository export and import is not required when making a modification to a schema.

For any transaction that includes the above special predicate:

- Read-only (explicit or inferred) statements can be deleted
- New explicit statements are marked as read-only
- New inferred statements are marked:
 - Read-only if all the premises that fired the rule are read-only
 - Normal otherwise

Predefined Rule-Sets

There are a number of pre-defined rule-sets provided with OWLIM-SE that cover various well known knowledge representation formalisms. The following table gives the details:

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



Note that all rule-sets do not support data-type reasoning, which is the main reason that OWL-Horst is not the same as pD*. The rule-set to be used for a specific repository is defined through the **ruleset** parameter. There are optimized versions of all rule-sets that avoid some little used inferences.

OWL2 QL non-conformance

The implementation of OWL2 QL is non-conformant with the W3C OWL2 profiles recommendation [31] as shown in Table 3:

Conformant behaviour	Implemented behaviour
----------------------	-----------------------

Given a list of disjoint (data or object) properties and an entity that is related with these properties to objects {a, b, c, d,...} then infer an owl:AllDifferent restriction on an anonymous list of these objects.	For each pair {p, q} (p != q) of disjoint (data or object) properties then infer the triple: p owl:propertyDisjointWith q Which is more likely to be useful for query answering.
For each class C in the knowledge base infer the existence of an anonymous class that is the union of a list of classes containing only C.	Not supported. Even if this infinite expansion were possible in a forward-chaining rule-based implementation, the resulting statements are of no use during query evaluation.
If a instance of C1, and b instance of C2, and C1 and C2 disjoint then infer: a owl:differentFrom b	Impractical for knowledge bases with many members of pairs of disjoint classes, e.g. Wordnet. Instead this is implemented as a consistency check: If x instance of C1 and C2, and C1 and C2 disjoint then inconsistent.

Custom Rule-Sets

OWLIM has an internal rule compiler that can be configured with a custom set of inference rules and axioms. The user may define a custom rule-set (see 'The Rule Language' on page) in a **.pie** file (e.g. **MySemantics.pie**). The easiest way to create a custom rule-set is to start modifying one of the **.pie** files that were used to build the precompiled rule-sets. All of these are provided as part of the OWLIM-SE distribution.

If the code generation or compilation cannot be completed successfully, a Java exception is thrown with an indication of the problem. It will state either the **Id** of the rule or the complete line from the source file where the problem is located. Line information is not preserved during the parsing of the rule file.

The user should specify the custom rule-set via the **ruleset** configuration parameter. The value of the **ruleset** parameter is interpreted as a filename and '.pie' is appended when not present. This file is processed to create Java source code that is compiled using the compiler from the Java Development Kit (JDK). The compiler is invoked using the mechanism provided by the JDK version 1.6 (or later). Therefore a prerequisite for using custom rule-sets is that the Java Virtual Machine (JVM) from a JDK version 1.6 (or later) is used to run the application. If all goes well, the class is loaded dynamically and instantiated for further use by OWLIM-SE during inference. The intermediate files are created in the folder that is pointed by the **java.io.tmpdir** system property. The JVM should have sufficient rights to read and write to this directory.



An important issue with custom rule sets is that at least one of the following RDF(S) resources be specified/mentioned within an axiomatic triple or a rule: **rdf:type**, **rdfs:range**, **rdfs:domain**, **rdfs:subClassOf**, **rdfs:class** and **rdfs:subPropertyOf**. These are used in specific implementations related to basic RDF support and are hard-coded into OWLIM-SE.



Due to some of the optimisation techniques used in OWLIM-SE, the set of rules in a custom rule set require at least one of the rules to derive the following triple patterns: **x rdf:type y**, **x rdfs:subClassOf y** and **x rdfs:subPropertyOf y** (the actual variable names do not matter).

Performance Optimizations in RDFS and OWL Support

There are several features in the RDFS and OWL specifications that result in rather inefficient entailment rules and axioms, which can have a significant impact on the performance of a reasoning engine. Such examples are:

- The consequence **X rdf:type rdfs:Resource** for each URI node in the RDF graph;
- The system should be able to infer that URIs are classes and properties if they appear in schema-defining statements like **X rdfs:subClassOf Y** and **X rdfs:subPropertyOf Y**;
- The individual equality property in OWL is reflexive, i.e. the statement **X owl:sameAs X** holds for every OWL individual;
- All OWL classes are sub-classes of **owl:Thing** and for all individuals **X rdf:type owl:Thing** should hold;
- A class is inferred as being a **rdfs:Class** whenever an instance of the class is defined with **I rdf:type C**.

Although the above inferences are correct and important for the completeness of the formal semantics, users rarely execute queries whose results are affected by the existence of such statements. Moreover, these inferences generate so many inferred statements that performance and scalability can be severely degraded.

For this reason, optimized versions of standard rule-sets are provided. These have '-optimized' appended to the rule-set name, e.g. **owl-horst-optimized**, and use the optimizations listed in Table 4.

Optimization	Affects
Remove axiomatic triples	<any> <any> <rdfs:Resource> <rdfs:Resource> <any> <any> <any> <rdfs:domain> <rdf:Property> <any> <rdfs:range> <rdf:Property> <owl:sameAs> <rdf:type> <owl:SymmetricProperty> <owl:sameAs> <rdf:type> <owl:TransitiveProperty>

Remove rule conclusions	<any> <any> <rdfs:Resource>
Remove rule constraints	[Constraint <variable> != <rdfs:Resource>]

These optimization were previously achieved using the **partialRDFS** parameter, but are now achieved by using a previously optimized built-in rule-set, see the **ruleset** parameter in the [configuration section](#) for a complete list.

owl:sameAs Optimization

The performance of a OWLIM-SE repository is greatly improved with a specific optimisation that allows it to handle `owl:sameAs` statements efficiently. `owl:sameAs` is an OWL predicate that declares that two different URIs identify one and the same resource. Most often, it is used to align different identifiers of the same real-world entity used in different data sources. For example, in DBpedia, the URI of Vienna is <http://dbpedia.org/page/Vienna>, while in Geonames it is <http://sws.geonames.org/2761369/>. DBpedia contains the statement

```
(S1) dbpedia:Vienna owl:sameAs geonames:2761369
```

which declares that the two URIs are equivalent. `owl:sameAs` is probably the most important OWL predicate when it comes to merging data from different data sources.

Following the formal definition of OWL (OWL2 RL, to be more specific), whenever two URIs are declared equivalent, all statements that involve one of the URIs should be "replicated" with the other URI at the same position. For instance, in Geonames, the city of Vienna is defined as part of <http://www.geonames.org/2761367/> (the first-order administrative division in Austria with the same name), which in turn, is part of Austria <http://www.geonames.org/2782113>:

```
(S2) geonames:2761369 gno:parentFeature geonames:2761367
(S3) geonames:2761367 gno:parentFeature geonames:2782113
```

Since `gno:parentFeature` is a transitive relationship, it will be inferred that the city of Vienna is also part of Austria:

```
(S4) geonames:2761369 gno:parentFeature geonames:2782113
```

Due to the semantics of `owl:sameAs` from (S1) it should also be inferred that statements (S2) and (S4) also hold for Vienna's DBpedia URI:

```
(S5) dbpedia:Vienna gno:parentFeature geonames:2761367
(S6) dbpedia:Vienna gno:parentFeature geonames:2782113
```

These implicit statements must hold no matter which one of the equivalent URIs is used, i.e. if a query is evaluated, the same results will be returned. When we consider that Austria, too, has an equivalent URI in DBpedia:

```
(S7) geonames:2782113 owl:sameAs dbpedia:Austria
```

we should also infer that:

```
(S8) dbpedia:Vienna gno:parentFeature dbpedia:Austria
(S9) geonames:2761369 gno:parentFeature dbpedia:Austria
(S10) geonames:2761367 gno:parentFeature dbpedia:Austria
```

In the above example, we had two alignment statements (S1 and S7), two statements carrying specific factual knowledge (S2 and S3), one statement inferred due to a transitive property (S4), and seven statements inferred as a result of `owl:sameAs` alignment (S5, S7, S8, S9, S10, as well as the inverse statements of S1 and S7). As we see, inference without `owl:sameAs` inflated the dataset by 25% (one new statement on top of 4 explicit), while the presence of the `owl:sameAs` statements increased the full closure by 175% (7 new statements). Considering that Vienna has a URI also in UMBEL, which is also declared equivalent to the one in DBpedia, the addition of one more explicit statement for this alignment, will cause the inference of 4 new implicit statements (duplicates of S1, S5, S6, and S8). Although this is a small example, it provides a indication about the performance implications of using `owl:sameAs` alignment statements from Linked Open Data. Also, because `owl:sameAs` is a transitive, reflexive, and symmetric relationship, for a set of N equivalent URIs N^2 (N squared) `owl:sameAs` statements will be generated for each pair of URIs (in reality there are not that many examples of large `owl:sameAs` equivalence classes). Thus, although `owl:sameAs` is useful for interlinking RDF datasets, its semantics causes considerable inflation of the number of implicit statements that should be considered during inference and query evaluation (either through forward- or through backward-chaining).

To overcome this problem, OWLIM-SE handles `owl:sameAs` in a specific manner. In its indices, each set of equivalent URIs (equivalence class with respect to `owl:sameAs`) is represented by a single super-node. This way, OWLIM-SE does not inflate the indices

and, at the same time, retains the ability to enumerate all statements that should be inferred using the equivalence during retrieval requests, i.e. during inference or query evaluation. Special care is taken to ensure that this optimisation does not hinder the ability to distinguish explicit from implicit statements.



The handling of `owl:sameAs` is technically a kind of backward chaining that occurs at query time, when equivalent URIs are enumerated and substituted in to query results. It should be noted that this will occur even when the 'empty' (no inference) rule-set is selected, i.e. even with no semantics selected `owl:sameAs` is still interpreted in a special way.

However, the `owl:sameAs` optimisation can be disabled completely using the `disable-sameAs` configuration parameter, see the configuration section for details.

OWLIM-SE Indexing Specifics

- [Comparison of OWLIM-SE and OWLIM-Lite](#)
- [Persistence Strategy](#)
- [Predicate Lists](#)
- [Context and Tripletset Indices](#)
- [Transaction Support](#)
- [Handling of Explicit and Implicit Statements](#)
 - [Retrieving Statements with the Sesame API](#)
 - [SPARQL Query Evaluation](#)

Comparison of OWLIM-SE and OWLIM-Lite

The major differences between OWLIM-SE and OWLIM-Lite are their performance and scalability. Both OWLIM editions deliver identical functionality for RDF storage, inference and query answering and they both implement Sesame's SAIL APIs, as discussed in section 4. This guarantees that all essential functions of a semantic repository are supported by OWLIM in a standard, consistent, and interoperable manner.

Compared to OWLIM-Lite, OWLIM-SE adds additional features and augments many aspects related to performance, which in most cases is a matter of special indexing strategies that allow more efficient retrieval. Functionally, the differences can be classified in to two groups:

- Do the same better: the corresponding feature does not allow the user to do more things with OWLIM-SE – it rather makes it work better in specific circumstances. This is the case with predicate lists (section 6.2) and the owl:sameAs optimisation (section 7.5);
- Do more: deliver a new type of functionality, which is not available in OWLIM-Lite. Such examples are RDF ranking and RDF priming (sections 10.1 and 10.4). Full-text search features (section 10.1) are also available in OWLIM-SE, although this could be seen as an enhancement since similar, but less powerful, behaviour is available by using regular expression constraints (which disregard tokenization and deliver different results).

In the 'do more' category, OWLIM-SE delivers functionality that is not exposed by the Sesame API. Typically, this is achieved with the use of special-purpose system predicates. One should be aware that using the 'do more' features will affect compatibility with other semantic repositories.

Persistence Strategy

OWLIM-SE stores all of its data (statements, indexes, entity pool, etc) in files in the configured storage directory, usually called 'storage'. The content and names of these files is not defined and is subject to change between versions. In general, the index structures used in OWLIM-SE are chosen and optimised to allow for efficient:

- handling of billions of facts under reasonable RAM constraints;
- query optimisation;
- transaction management;
- multi-threaded inference.

OWLIM-SE maintains two main indices on statements for use in inference and query evaluation, these are the predicate-object-subject (POS) index and the predicate-subject-object (PSO) index. There are many other additional data structures that are used to enable the efficient manipulation of RDF data, but these are not listed since these internal mechanisms cannot be configured. The following subsections describe several indexing options, which deliver considerable advantages for specific datasets, retrieval patterns and query loads. Most of these are switched off by default, thus the user should take the initiative to switch them on if necessary. Unless otherwise stated, OWLIM-SE allows one to switch indices on and off against an already populated repository; the repository should be shut down before the change of the configuration is specified. The next time the repository is started, OWLIM-SE will create or remove the corresponding index. In the case that the repository is already loaded with a large volume of data, switching on a new index can lead to considerable delays during initialisation – this is the time required for building the new index.



It is vitally important to shutdown repository connections properly to ensure that unwritten data is flushed to the file-system. OWLIM-SE includes mechanisms to prevent data loss in the case of unexpected termination of the process, however, recovery after unexpected termination may require additional time and data can be lost for reasons beyond OWLIM-SE's control, e.g. operating system caching.

Predicate Lists

Certain data-sets and certain kinds of query activities, for example queries that use wild-card patterns for predicates, benefit from another type of index called a 'predicate list'. This index maps from entities (subject or object) to their predicates. This index is not switched on by default (see **enablePredicateList** in section 8.5), because it is not always necessary. Indeed, for most datasets and query loads the performance of OWLIM-SE without such an index is good enough even with wild-card-predicate queries, and the overhead of maintaining this index are not justified. One should consider using this index for datasets that contain a very large number (~1000) different predicates.

Context and Tripletset Indices

There are two more optional indices that can be used to speed up query evaluation when searching statements via their context or tripletset identifier. These indices are the PCSOT and the PTSOC indices and can be switch on independently. See the **build-pcsot** and **build-ptsoc** parameters in section 8.5.

Transaction Support

Transaction support is exposed via Sesame's **RepositoryConnection** interface. The three methods of this interface that give the client control over when updates are committed to the repository are shown below:

Method	Effect
<code>void commit()</code>	Commits all updates that have been performed through this connection since the last commit or rollback operation.
<code>void rollback()</code>	Rolls back all updates that have been performed through this connection object since the last commit or rollback operation.
<code>void setAutoCommit(boolean autoCommit)</code>	Enables or disables auto-commit mode for the connection.

OWLIM-SE supports the so called 'read committed' transaction isolation level, well known to relational database management systems. It guarantees that changes will not impact query evaluation, before the entire transaction they are part of is successfully committed. It does not guarantee that execution of a single transaction is performed against a single state of the data in the repository. Regarding concurrency:

- multiple update/modification/write transactions can be initiated and stay open simultaneously, i.e. one transaction does not need to be committed in order to allow another transaction to complete;
- update transactions are processed internally in sequence, i.e. OWLIM processes the commits one after another;
- update transactions do not block read requests in any way, i.e. hundreds of SPARQL queries can be evaluated in parallel (the processing is properly multi-threaded) while update transactions are being handled on separate threads.

One should note that OWLIM performs materialization, making sure that all the statements which can be inferred from the current state of the repository are indexed and persisted (except for those compressed due to the **owl:sameAs** optimisation, described in section 7.5). When the commit method completes, all reasoning related activities related to the changes in the data introduced by the corresponding transaction will have already been performed.



An uncommitted transaction will not affect the 'view' of the repository through any connection, including the connection used to do the modification. This is perhaps not in keeping with most relational database implementations. However, committing a modification to a semantic repository involves considerably more work, specifically the computation of the changes to the inferred closure resulting from the addition or removal of explicit statements. This computation is only carried out at the point where the transaction is committed and so to consistent, neither the inferred statements nor the modified statements related to the transaction are 'visible'.

Handling of Explicit and Implicit Statements

As already described, OWLIM-SE applies the inference rules at load time in order to compute the full closure. Therefore a repository will contain some statements that are explicitly asserted and other statements that exist through implication. In most cases clients will not be concerned with the difference, however there are some scenarios when it is useful to work with only explicit or only implicit statements. The following sections describe how these two groups of statements can be isolated during programmatic statement retrieval using the Sesame API and during (SPARQL) query evaluation.

Retrieving Statements with the Sesame API

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

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

The method retrieves statements by 'triple pattern', where any or all of the subject, predicate and object parameters can be **null** to indicate 'wild cards'.

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

However, the Sesame API does not provide the means to enable the retrieval of implicit statements only. In order to allow clients to do this, OWLIM-SE allows the use of the special 'implicit' pseudo-graph (section 10.7.1) with this API, which can be passed as the context parameter. The following example shows how only implicit statements can be retrieved:

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

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

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

SPARQL Query Evaluation

OWLIM-SE also provides mechanisms to differentiate between explicit and implicit statements during query evaluation. This is achieved by associating statements with two pseudo-graphs (explicit and implicit) and using special system URIs to identify these graphs. Full details can be found in the [query behaviour](#) section.

OWLIM-SE Installation

This section is for users and system administrators that are unfamiliar with the OWLIM-SE semantic repository software. The information contained in the following pages should be enough to get started with OWLIM-SE, i.e. to install and configure the software so that repository instances can be created and used.

- [Usage scenarios](#)
- [Upgrading](#)
- [Easy install](#)
- [Longer install](#)
 - [Preparation](#)
 - [Installation](#)
 - [Step by step \(vanilla\)](#)
 - [Step by step \(Fedora linux\)](#)
 - [Step by step \(Windows\)](#)
- [Useful information](#)
 - [Tomcat](#)
 - [Sesame](#)
 - [Logging](#)
- [Common Problems](#)
 - [Can not copy OWLIM-SE jar file to the Sesame WEB-INF/lib directory.](#)
 - [Can not connect the Sesame console to the local Sesame server at `http://localhost:8080/openrdf-sesame`](#)
 - [Can not create a OWLIM-SE repository using the Sesame console](#)
 - [Can not create an OWLIM-SE repository, the Sesame console says unknown Sail type - owl:Sail](#)
 - [A long error message/Java exception is given when trying to query a new repository instance.](#)
 - [Cannot use my custom rule file \(pie file\), an exception occurred](#)
 - [Sesame Workbench starts, but gives memory error on the 'explore' and 'query' menus.](#)
- [Installing OWLIM-SE with Jena](#)
 - [Required software](#)
 - [Description of the OWLIM-SE Jena adapter](#)
 - [Instantiate Jena adapter using a SailRepository](#)
 - [Instantiate OWLIM-SE adapter using the provided Assembler](#)
 - [Using OWLIM-SE with the Joseki server](#)

Usage scenarios

OWLIM-SE is packaged as a Storage and Inference Layer (SAIL) for the [Sesame RDF framework](#). OWLIM-SE can be used in two different ways:

- **Library:** One approach is to integrate OWLIM in an application using it as a library. An example of doing this is given in the 'getting-started' folder of the OWLIM-SE distribution zip file. Users intending to use OWLIM-SE in this way do not need to install the Sesame Web applications (Server and Workbench) and need only set a few environment variables as detailed in the [configuration section of the OWLIM-SE user guide](#).
- **Server:** Those users wishing to use OWLIM-SE via the Sesame Server, with or without the Sesame Workbench, should proceed with one of the two approaches detailed in this section. This will cover installing Sesame and integrating OWLIM-SE in to the Sesame framework, so that OWLIM-SE can be used as a plug-in repository type. This approach involves deploying Sesame/OWLIM-SE via a servlet server, such as Tomcat. The Sesame installation comes with a useful 'workbench' Web application that can be used for many administration tasks.

The rest of this section gives detailed step by step instructions for installing and configuring Sesame and OWLIM-SE, leading up to the creation of an OWLIM-SE repository using the Sesame console (a command line script). For more information please check the "doc" folder of the OWLIM-SE distribution zip file.

Upgrading

In most cases, upgrading OWLIM-SE can be achieved simply by replacing the OWLIM and bundled 3rd party jar files. The required version of the Sesame framework for recent OWLIM versions is given in the following table:

OWLIM version	Required Sesame version
OWLIM-SE 4.3	Sesame 2.6
OWLIM-SE 4.2	Sesame 2.5
OWLIM-SE 4.1	Sesame 2.4
OWLIM-SE 4.0	Sesame 2.4
BigOWLIM 3.5	Sesame 2.3
BigOWLIM 3.4	Sesame 2.3

BigOWLIM 3.3	Sesame 2.3
BigOWLIM 3.2	Sesame 2.3
BigOWLIM 3.1	Sesame 2.2

Typically, the binary format of the storage files for different versions of BigOWLIM/OWLIM-SE are not compatible. However, storage files created with version 3.1 and upwards will be automatically upgraded when using a later version of OWLIM. However, this is a one-way process and no downgrade facility is provided.



When upgrading to a later version of OWLIM it is vitally important to make a back-up of your storage files before running the new software, because it will be impossible to revert to a previous version of OWLIM otherwise.

Easy install

OWLIM will normally be built against the latest version of the [Aduna OpenRDF Sesame framework](#). If using OWLIM with this version then the easiest way to deploy an OWLIM server instance is to copy the re-packaged Sesame Web applications (.war files) in to the Tomcat webapps directory. These can be found in the `sesame_owlim` directory in the OWLIM distribution ZIP file. These Web applications have been modified as follows:

- **openrdf-sesame.war:** The Sesame server application. All OWLIM-SE and 3rd party jar files have been added to this package, so that it can be used immediately after deployment.
- **openrdf-workbench.war:** The OWLIM-SE jar, XSLT configuration code and an OWLIM-SE Turtle configuration file have been added, so that new OWLIM-SE instances can be created using the Workbench without the need to run the command line console application.

After deployment, an OWLIM-SE instance can be created using the workbench application as follows:

- Start a browser and go to the Workbench Web application using a URL of this form: <http://localhost:8080/openrdf-workbench/> - substituting `localhost` and the `8080` port number as appropriate.
- Click 'change server' if necessary to select a Sesame server instance if it is running on a different machine. The server URL will be of this form: <http://localhost:8080/openrdf-sesame>
- After selecting a server, click 'New repository'
- Select 'OWLIM-SE' from the drop-down and enter the repository ID and description. Then click 'next'.
- Fill in the fields as required (referring to the owl-configurator spreadsheet as necessary) and click 'create' - if you have an OWLIM-SE license file, its location should be specified in the license field.
- A message should be displayed that gives details of the new repository and this should also appear in the repository list (click 'repositories' to see this).

Longer install

This longer install process should be used when a different or modified version of Sesame is being used. In this case, all the OWLIM-SE files need to be installed alongside Sesame in a step-by-step process as follows.

Preparation

A suitable application server must be installed. The examples in this guide assume [Apache Tomcat](#) version 6.x is used. Importantly, the user must have sufficient privileges to write to the Tomcat `webapps` directory and to stop and start the Tomcat daemon.

An [OWLIM-SE](#) distribution is required. This is published as a zip file and includes the [Lucene](#) (core) full-text search utility jar file. The examples in this guide use version 4.3.

If desired, the core version of Lucene can be replaced with the full Lucene jar that is available for download from the [Lucene website](#).

The [Aduna OpenRDF Sesame framework version 2.4](#) is required for the longer installation. The examples in this guide use version 2.6.



Another application server can be used instead of Tomcat. However, the rest of this guide will describe the installation of OWLIM-SE with this software.

No part of this guide is intended to supersede the documentation published with these 3rd party software components and the reader is strongly advised to familiarise himself/herself with these.

Installation

Assuming that an instance of Tomcat is available, the installation proceeds as follows:

- Install and deploy Sesame
- Install OWLIM-SE and deploy OWLIM-SE and other 3rd party jar files (including Lucene-core)
- Add the OWLIM-SE repository template file to the Sesame console configuration
- Use the Sesame console to create a new repository.

- Set up a license file if you have one.

Step by step (vanilla)

- Locate the Tomcat webapps directory, hereafter called [WEBAPPS].
- Unzip the Sesame 2 distribution zip file. The Sesame 'root' directory will be called [SESAME].
- Unzip the OWLIM-SE distribution. The top level unzipped directory will be called [OWLIMSE].
- Deploy the Sesame Web applications by copying the war files in [SESAME]/war to [WEBAPPS].



Alternatively, use the Tomcat manager interface at <http://localhost:8080/manager/html> to deploy the Web applications. If this approach is used then ensure that a manager user account exists. The following lines should be added to: [tomcat-directory]/conf/tomcat-users.xml substituting your own password:

```
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="pass" roles="manager"/>
</tomcat-users>
```

- Copy the SL4J logger libraries from the Sesame Server to the Sesame Workbench application, i.e. from [WEBAPPS]/openrdf-sesame/WEB-INF/lib/logback-* to [WEBAPPS]/openrdf-workbench/WEB-INF/lib
- Restart Tomcat
- Run the Sesame console once to create its configuration directory by executing [SESAME]/bin/console.sh (.bat) and then type quit . at the command prompt.
- Locate the data directory here:

```
$HOME/.aduna/openrdf-sesame-console* (linux/unix)
"%APPDATA%\Aduna\OpenRDF Sesame console" (Windows)
```

- Create the templates folder here and copy the OWLIM-SE repository template to it. This file is found here: [OWLIMSE]/templates/owlim-se.ttl
- Copy the OWLIM-SE jar file to the Sesame console and Sesame server, i.e. from [OWLIMSE]/lib to [SESAME]/lib and [WEBAPPS]/openrdf-sesame/WEB-INF/lib
- Copy the Lucene and other 3rd party jars to the Sesame server, i.e. from [OWLIMSE]/ext to [WEBAPPS]/openrdf-sesame/WEB-INF/lib
- Restart Tomcat
- Run the Sesame console [SESAME]/bin/console.sh (.bat)
- Connect to the local Sesame server with: connect <http://localhost:8080/openrdf-sesame> .
- Create a new repository: create owlim-se.
- Accept all the default values and then exit the console: quit .

The status of the new repository can be checked using the Sesame workbench. Type the following URL in to a browser:

```
http://localhost:8080/openrdf-workbench
```

Step by step (Fedora linux)

- Login as root and ensure that tomcat is running:

```
service tomcat6 status
```

- Unzip the Sesame 2 installation:

```
unzip openrdf-sesame-2.*-sdk.zip
```

- Deploy the Sesame Web applications (Sesame server and workbench):

```
cp openrdf-sesame-2.*war/*.war /var/lib/tomcat6/webapps/
```

- Copy the logback logging libraries:

```
cp /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/logback-*
/var/lib/tomcat6/webapps/openrdf-workbench/WEB-INF/lib/
```

- Restart Tomcat:

```
service tomcat6 restart
```

- Run the Sesame console once to create the data/configuration directory:

```
openrdf-sesame-2.*/bin/console.sh
```

- And exit:

```
quit .
```

- Create the Sesame console's templates directory:

```
mkdir $HOME/.aduna/openrdf-sesame-console/templates
```

- Unzip the OWLIM-SE distribution:

```
unzip owl-se-4.*.zip
```

- Copy the OWLIM-SE repository template file to the Sesame console templates directory:

```
cp owl-se-4*/templates/owlim-se.ttl $HOME/.aduna/openrdf-sesame-console/templates/
```

- Copy the OWLIM-SE jar file to the Sesame console lib directory:

```
cp owl-se-4*/lib/owlim-se*.jar openrdf-sesame-2.*/lib/
```

- Copy the OWLIM-SE jar file to the Sesame server lib directory:

```
cp owl-se-4*/lib/owlim-se*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/
```

- Copy the Lucene jar file the other 3rd party jar files to the Sesame server Web application:

```
cp owl-se-4*/ext/log4j*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/  
cp owl-se-4*/ext/lucene*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/  
cp owl-se-4*/ext/jsi*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/  
cp owl-se-4*/ext/onto-crypto*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/  
cp owl-se-4*/ext/sil*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/  
cp owl-se-4*/ext/trove4j*.jar /var/lib/tomcat6/webapps/openrdf-sesame/WEB-INF/lib/
```

- Restart Tomcat:

```
service tomcat6 restart
```

- Run the Sesame console to create a new repository:

```
openrdf-sesame-2.*/bin/console.sh
```

- Connect to the Sesame server:

```
connect http://localhost:8080/openrdf-sesame .
```

- Create a new repository:

```
create owl-se .
```

- At this point values for a number of parameters are asked for, but all the defaults can be accepted by pressing <return> for each question. When finished, the message 'Repository created' should be displayed.

- Exit the console:

```
quit.
```

- The status of the new repository can be checked using the Sesame workbench. Type the following URL in to a browser:

```
http://localhost:8080/openrdf-workbench
```

- Alternatively, execute a test query against the Sesame Server's SPARQL endpoint:

```
curl -H "Accept: application/sparql-results+json"  
'http://localhost:8080/openrdf-sesame/repositories/owlim-se-test?infer=false&queryLn=sparql6
```

Step by step (Windows)

- Unzip the Sesame 2 installation (note that this might require WinZIP, WinRAR or some other utility):

```
unzip openrdf-sesame-2.6.0-sdk.zip
```

- Deploy the Sesame Web applications (Sesame server and workbench):

```
copy openrdf-sesame-2.6.0\war\*.war "C:\Program Files\Apache Software Foundation\Tomcat  
6.0\webapps"
```

- Restart Tomcat using the Tomcat monitor icon in the system tray or from the Start menu
- Run the Sesame console once to create the data/configuration directory:

```
openrdf-sesame-2.6.0\bin\console.bat
```

- Then exit immediately:

```
quit .
```

- Create the Sesame console's templates directory:

```
mkdir "%APPDATA%\Aduna\OpenRDF Sesame console\templates"
```

- Unzip the OWLIM-SE distribution:

```
unzip owl-se-4*.zip
```

- Copy the OWLIM-SE repository template file to the Sesame console templates directory:

```
copy owl-se-4.3.4220\templates\owlim-se.ttl "%APPDATA%\Aduna\OpenRDF Sesame  
console\templates"
```

- Copy the OWLIM-SE jar file to the Sesame console and Sesame server lib directories:

```
copy owl-se-4.3.4220\lib\owlim-se-4*.jar openrdf-sesame-2.6.0\lib  
copy owl-se-4.3.4220\lib\owlim-se-4*.jar  
"C:\Program Files\Apache Software Foundation\Tomcat  
6.0\webapps\openrdf-sesame\WEB-INF\lib"
```

- Copy the Lucene and other 3rd party jar files to the Sesame server Web application:

```
copy owl-lite-4.3.4220\ext\log4j*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
copy owl-lite-4.3.4220\ext\lucene*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
copy owl-lite-4.3.4220\ext\jsi*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
copy owl-lite-4.3.4220\ext\onto-crypto*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
copy owl-lite-4.3.4220\ext\sil*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
copy owl-lite-4.3.4220\ext\trove*.jar
"C:\Program Files\Apache Software Foundation\Tomcat
6.0\webapps\openrdf-sesame\WEB-INF\lib"
```

- Restart Tomcat using the Tomcat monitor icon in the system tray or from the Start menu
- Run the Sesame console to create a new repository:

```
openrdf-sesame-2.6.0\bin\console.bat
```

- Connect to the Sesame server:

```
connect http://localhost:8080/openrdf-sesame.
```

- Create a new repository:

```
create owl-se.
```

- At this point values for a number of parameters are asked for, but all the defaults can be accepted pressing <return> for each question. When finished, the message 'Repository created' should be displayed, so exit the console:

```
quit.
```

- The status of the new repository can be checked using the Sesame workbench. Type the following URL in to a browser:

```
http://localhost:8080/openrdf-workbench/
```

Useful information

Tomcat

When running on Fedora linux, Tomcat uses the following directories:

Contents	Directory
Log files	/var/log/tomcat6/
Web applications	/var/lib/tomcat6/webapps
Sesame configuration and repository data files	/usr/share/tomcat6/.aduna

On a machine running Windows, Tomcat uses the following directories:

Contents	Directory
Log files	C:\Program Files\Apache Software Foundation\Tomcat 6.0\logs
Web applications	C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps

Sesame configuration and repository data files	C:\Users\<username>\AppData\Roaming\Aduna
--	---

Sesame

The Sesame framework is published as a zip file that is installed simply by unzipping it in to the target directory. This directory is called [SESAME_INSTALL] in this guide.

When running, Sesame stores configuration and repository data in the following directory:

\$HOME/.aduna/	On linux/unix operating systems.
%APPDATA%\Aduna	On Windows machines. Note that if Sesame is running as a service then the full path will be: C:\Documents and Settings\LocalService\Application Data\Aduna

This directory is called [ADUNA_DATA] elsewhere in this guide. It can be overridden by setting a Java system property. When using Tomcat, this can be achieved by setting the JAVA_OPTS environment variable, for example (linux and Windows respectively):

```
export JAVA_OPTS='-Dinfo.aduna.platform.appdata.basedir=/other/dir/'
set JAVA_OPTS=-Dinfo.aduna.platform.appdata.basedir=\other\dir\
```

Note that setting this environment variable must be done by the same user that runs the Tomcat server. This may not be the case when using 'sudo' to set up/start Tomcat, for instance when using port numbers below 1024 when it is necessary to run Tomcat with root privileges.

The Sesame 2 framework uses a special SYSTEM repository to store its configuration data. Do not store anything else in this repository. The workbench Web application can be used for many administrative tasks, but it is still experimental. The easiest method to create new repositories is to use the Sesame Console script, which can be found in the bin directory: [SESAME_INSTALL]/bin/console.bat (.sh for linux/unix)



All commands entered using the console must end in a dot '.', e.g. 'help.'

Logging

The Sesame server and workbench use logback for logging. The Sesame server logs to:

```
[ADUNA_DATA]/openrdf-sesame/logs/main.log
```

And the Workbench logs to:

```
[ADUNA_DATA]/openrdf-workbench/logs/main.log
```

The default log level is INFO, but this can be adjusted after the first run by editing:

```
[ADUNA_DATA]/openrdf-sesame/conf/logback.xml
```

There is a note in the Sesame documentation that the simple logging framework for java (slf4j) is used, which implies that a bridge jar for the logging framework OWLIM uses is also required, e.g. SLF4J-to-log4j.jar

Common Problems

Can not copy OWLIM-SE jar file to the Sesame WEB-INF/lib directory.

This directory will not exist until the Sesame war files have been deployed to the [WEBAPPS] directory AND Tomcat is running. If the war files have been deployed, but the directory does not exist, try restarting Tomcat.

Can not connect the Sesame console to the local Sesame server at <http://localhost:8080/openrdf-sesame>

Make sure that the Sesame war files have been deployed and that Tomcat is running. Restart Tomcat if necessary.

Can not create a OWLIM-SE repository using the Sesame console

Make sure that the repository template file [OWLIMSE]/templates/owlim-se.ttl has been copied to the 'templates' subdirectory of the

Sesame console's data directory.

Can not create an OWLIM-SE repository, the Sesame console says unknown Sail type – owl:Sail

The Sesame console can not find the OWLIM-SE jar file. Make sure it was copied from [OWLIMSE]/lib to the Sesame installation folder here: [SESAME]/lib

A long error message/Java exception is given when trying to query a new repository instance.

The Lucene jar file must be copied from the OWLIM-SE distribution to the Sesame server's WEB-INF/lib directory. Restart Tomcat if necessary.

Cannot use my custom rule file (pie file), an exception occurred

The tools.jar file from the Java Development Kit (JDK) must be on the classpath or alternatively copied to Sesame server's WEB-INF/lib directory.

Sesame Workbench starts, but gives memory error on the 'explore' and 'query' menus.

The maximum heap space must be increased, i.e. Tomcat's Java virtual machine must be allowed to allocate more memory. This can be done by setting the environment variable 'CATALINA_OPTS' to include the desired value, e.g. -Xmx1024m

Installing OWLIM-SE with Jena

This section is for users and system administrators that are unfamiliar with the OWLIM-SE Jena adapter. The information contained in the following pages should be enough to get started with this software, i.e. to install and configure an OWLIM-SE repository and access it via the Jena API or Jena compatible tools.



The Jena adapter allows OWLIM-SE to be used with local repositories, NOT those shared via the Sesame HTTP server. If you want to use OWLIM-SE remotely, consider using the Joseki server as described below.

Required software

This guide covers installing and configuring the Jena adapter for [OWLIM-SE](#). Required software for this comprises:

- [Jena](#) version 2.6 (tested with version 2.6.3)
- [ARQ](#) (tested with version 2.8.6)
- [OWLIM-SE](#) v3.5

No part of this guide is intended to supersede the documentation published with these software components and the reader is strongly advised to familiarise himself/herself with these.

Description of the OWLIM-SE Jena adapter

The OWLIM-SE Jena adapter is essentially an implementation of the [Jena DatasetGraph](#) interface that provides access to individual triples managed by an OWLIM-SE repository through the Sesame API interfaces.

It is not a general purpose Sesame adapter and cannot be used to access any Sesame compatible repository, because it utilises an internal OWLIM-SE API to provide more efficient methods for processing RDF data and evaluating queries.

The adapter comes with its own implementation of a [Jena](#) 'assembler' factory to make it easier to instantiate and use with those related parts of the Jena framework, although one can instantiate an adapter directly by providing an instance of a Sesame **SailRepository** (an OWLIM-SE **OWLIMRepository** implementation). Query evaluation is controlled by the ARQ engine, but specific parts of a Query (mostly batches of statement patterns) are evaluated natively through a modified **StageGenerator** plugged into the Jena runtime framework for efficiency. This also avoids unnecessary cross-api data transformations during query evaluation.

Instantiate Jena adapter using a SailRepository

In this approach, an OWLIM-SE repository is first created and wrapped in a Sesame **SailRepository**. Then a connection to it is used to instantiate the adapter class **SesameDataset**. The following example helps to clarify:

```

import com.ontotext.trree.OwlimSchemaRepository;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.RepositoryConnection;
import com.ontotext.jena.SesameDataset;

...

OwlimSchemaRepository schema = new OwlimSchemaRepository();

// set the data folder where OWLIM-SE will persist its data
schema.setDataDir(new File("./local-sotrage"));

// configure OWLIM-SE with some parameters
schema.setParameter("storage-folder", ".");
schema.setParameter("repository-type", "file-repository");
schema.setParameter("ruleset", "rdfs");

// wrap it into a Sesame SailRepository
SailRepository repository = new SailRepository(schema);

// initialize
repository.initialize();
RepositoryConnection connection = repository.getConnection();

// finally, create the DatasetGraph instance
SesameDataset dataset = new SesameDataset(connection);

```

From now on the `SesameDataset` object can be used through the Jena API as regular `Dataset`, e.g. to add some data into it one could something like the following:

```

Model model = ModelFactory.createModelForGraph(dataset.getDefaultGraph());
Resource r1 = model.createResource("http://example.org/book#1") ;
Resource r2 = model.createResource("http://example.org/book#2") ;

r1.addProperty(DC.title, "SPARQL - the book")
  .addProperty(DC.description, "A book about SPARQL") ;

r2.addProperty(DC.title, "Advanced techniques for SPARQL") ;

```

and can also be used to evaluate queries through the ARQ engine:

```

// Query string.
String queryString = "PREFIX dc: <" + DC.getURI() + "> " +
    "SELECT ?title WHERE {?x dc:title ?title . }";

Query query = QueryFactory.create(queryString);

// Create a single execution of this query, apply to a model
// which is wrapped up as a QueryExecution and then fetch the results
QueryExecution qexec = QueryExecutionFactory.create(query, dataset.asDataset());
try {
    // Assumption: it's a SELECT query.
    ResultSet rs = qexec.execSelect();
    // The order of results is undefined.
    for (; rs.hasNext(); ) {
        QuerySolution rb = rs.nextSolution();
        for (Iterator<String> iter = rb.varNames(); iter.hasNext(); ) {
            String name = iter.next();
            RDFNode x = rb.get(name);
            if (x.isLiteral()) {
                Literal titleStr = (Literal) x;
                System.out.print(name + "=" + titleStr + "\t");
            } else if (x.isURIResource()) {
                Resource res = (Resource) x;
                System.out.print(name + "=" + res.getURI() + "\t");
            }
            else
                System.out.print(name + "=" + x.toString() + "\t");
        }
        System.out.println();
    }
}
catch( Exception e ) {
    System.out.println( "Exception occurred: " + e );
}
finally {
    // QueryExecution objects should be closed to free any system resources
    qexec.close();
}

```

Instantiate OWLIM-SE adapter using the provided Assembler

Another approach is to use the Jena assemblers infrastructure to instantiate an OWLIM-SE Jena adapter. For this purpose the required configuration must be stored in some valid RDF serialization format and its contents read into a Jena model. Then the assembler can be invoked to get an instance of the Jena adapter. The following example specifies an adapter instance in N3 format.

```

@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ja:       <http://jena.hpl.hp.com/2005/11/Assembler#> .
@prefix otjena:   <http://www.ontotext.com/jena/> .

@prefix :         <#> .

[] ja:loadClass "com.ontotext.jena.SesameVocab" .
otjena:SesameDataset rdfs:subClassOf ja:Object .
otjena:SesameDataset ja:assembler "com.ontotext.jena.SesameAssembler" .
<#dataset> rdf:type otjena:SesameDataset ;
            otjena:datasetParam "./location" .

```

The `ja:loadClass*` statements ensure that the OWLIM-SE Jena adapter factory class file(s) are initialized and plugged into the Jena framework prior to being invoked. Then **#dataset** description tells the Jena framework to expect instances of **otjena:SesameDataset** to be created by this factory. The following example uses such a description stored in the file **owlimbridge.n3** to get an instance of the Jena adapter:

```

Model spec = FileManager.get().loadModel( "owlimbridge.n3" );
Resource root = spec.createResource( spec.expandPrefix( ":dataset" ) );
DataSource datasource = (DataSource)Assembler.general.open( root );
DatasetGraph dataset = datasource.asDatasetGraph();

```

After this, the adapter is ready to be used, for example to evaluate some query through the ARQ engine using the same approach described a few paragraphs above.

Using OWLIM-SE with the Joseki server

In order to use a OWLIM-SE repository with the Joseki server one needs only to configure it as a dataset so that the Jena assembler framework is able to instantiate it. An example Joseki configuration file that makes use of such a dataset description could look like the following. First we describe a service that hosts the dataset:

```
<#service1>
  rdf:type          joseki:Service ;
  rdfs:label        "service point" ;
  joseki:dataset    otjena:bridge ;
  joseki:serviceRef  "sparql" ;
  joseki:processor   joseki:ProcessorSPARQL ;
  .
```

Then we describe that dataset:

```
[] ja:loadClass "com.ontotext.jena.SesameVocab" .
otjena:DatasetSesame rdfs:subClassOf ja:RDFDataset .
otjena:bridge rdf:type otjena:DatasetSesame ;
  rdfs:label "OWLIM-SE repository" ;
  otjena:datasetParam "./location" .
```


OWLIM-SE Configuration

This section gives an overview of configuring an OWLIM-SE repository. Also covered are the contents of the OWLIM-SE distribution and a description of the 'getting-started' application that is included. This sample application serves as an example for integrating OWLIM-SE in to other systems. For a detailed step-by-step guide for installing and setting-up OWLIM-SE, see the [installation section](#).

- [Contents of the Distribution Package](#)
- [Getting-Started Application](#)
 - [Wordnet example](#)
- [Configuration](#)
 - [Sample Configuration](#)
- [Memory Requirements](#)
 - [Cache Memory Configuration](#)
 - [Parameters](#)
 - [Memory Distribution](#)
 - [Backward Compatibility](#)
- [Configuration Parameters](#)
- [Performance analytics](#)

Contents of the Distribution Package

The OWLIM-SE distribution zip file includes the following folders:

Folder	Contents
doc	User guide, quick start guide and the OWLIM primer
ext	All required third party libraries. The Sesame 2 installation can be downloaded separately from http://www.openrdf.org/ . The folder also contains a copy of Lehigh University Benchmark library (lubm.jar), JUnit library (junit.jar) necessary for executing inference tests, the simple logging framework for Java jar files and the lucene full text search jar
getting-started	An example application that uses OWLIM, with all the necessary auxiliary files and folders, see section 8.2
lib	Contains the binary executable version of OWLIM-SE as a JAR (Java library) file
lubm	Scripts and configuration files to run the LUBM [16] benchmarks documented in section 11.
templates	Contains an OWLIM-SE repository template file (.ttl) used by the Sesame 2 framework for creating new repositories.

The distribution contains the following files in the root directory of the zip file:

File	Description
*.pie	Rule files containing definitions of the built-in rule-sets, see section 7.1.2.
OWLIM-SE_license_agreement_xx.pdf	The license under which OWLIM-SE is published.
owlim-se-configurator.xls	A useful memory requirement calculator and configuration tool that can be used to calculate the correct Java heap size, memory allocation and various other configuration parameters. Command line and turtle configurations are generated. Instructions for using this spreadsheet are given on the first page.
setvars.cmd setvars.sh	Scripts (Windows and Linux) that define several environment variables used by the scripts that run the test cases and getting-started application. It should be edited for each installation, as it determines the Java virtual machine to be started, the path to all the relevant JAR files, including those of OWLIM-SE and Sesame

Getting-Started Application

The OWLIM-SE distribution comes with a sample application that can be used as a template for building applications that interact with an OWLIM-SE repository. The source code of this application performs a sequence of typical operations: initialisation of the repository, uploading statements, executing queries and obtaining results, deleting statements, etc. This application template comes with:

- Source code and compiled class files
- Sample ontology and data files
- Sesame repository template file
- Scripts which invoke the application

An easy way to set up an application to use OWLIM-SE is to copy the `getting-started` folder and modify the contents as necessary.

There follows a short description on how the getting started application is organised and what the sample code does. The easiest way of getting a good understanding of it is to read the source code of the `GettingStarted` class, located in the `src` folder - the code is extensively commented.

The program accepts the following command line parameters:

Parameter	Description	Default
<code>context</code>	If not specified, statements loaded are given the context of the file URL from which the statements were loaded. If specified (<code>context=URI</code>), then all statements loaded are given this URI for the context. If an empty context is used (<code>context=</code>) then all statements loaded have no context (default graph)	<none>
<code>config</code>	Specifies the repository description file to use to create a repository. Configuration options specified in this file are explained in section 8.5.	<code>./owlim.ttl</code>
<code>flush</code>	Indicates whether repository updates should be flushed to disk after every commit	false
<code>preload</code>	Specifies the folder or file containing RDF data that is loaded automatically when the program starts. If the parameter value specifies a folder then it is searched recursively for all files that contain RDF data.	<code>./preload</code>
<code>queryfile</code>	Specifies the file containing queries that are to be executed. The files can contain queries in any format supported by Sesame.	<code>./queries/sample.sparql</code>
<code>repository</code>	The repository ID identifying the repository described in the configuration file specified by the <code>config</code> parameter.	owlim
<code>showresults</code>	Specifies whether the results from queries will be displayed or not.	true
<code>showstats</code>	Indicates whether to show initialisation statistics after loading the selected data files	false
<code>updates</code>	Specifies whether the statement insertion and deletion step is performed.	false
<code>url</code>	Indicates whether to connect to the given remote repository URL - overrides <code>config</code> and <code>repository</code> parameters	<none>

To run the program, use the `example.cmd / example.sh` script. This script requires that the `JAVA_HOME` environment variable has been set. Alternatively, it can be set directly by editing the `setvars.cmd / setvars.sh` script in the root folder of the OWLIM-SE software distribution. If the program is modified to use a custom rule set, then `JAVA_HOME` must point to the Java Runtime Environment (JRE) of a Java Development Kit (JDK) version 1.6 or later. This is so that the new mechanism for locating the Java compiler can be used.

With the example set up, OWLIM-SE loads two ontologies at start up as specified by the `imports` parameter in the repository configuration file, i.e. `owlim.ttl`. These ontologies are `./ontology/owl.rdfs` and `./ontology/example.rdfs`. The sample program then loads any other ontologies that it finds in the `preload` folder. When start up is complete, the program outputs some statistics and lists the namespaces found.

The next step is to load the specified query file and to execute the queries that they contain. Some example query files are included in the `queries` folder. The files can contain several queries where each query starts with an identifier, enclosed in square brackets `^[` and `]` on a single line; everything between two subsequent query identifiers is treated as a SeRQL or SPARQL query and is evaluated against the contents of the prepared repository. You may use also the `#` sign as a single line comment, so each line starting with `#` will be ignored. Syntax overview:

```
#some comment
^[queryid1]
<query line1>
<query line2>
...
<query lineN>
#some other comment
^[nextqueryid]
<query line1>
...
<EOF>
```

The queries are always evaluated, but the results are output only if the `showresults` parameter is set to `true`.

Furthermore, the sample application updates the contents of the repository by inserting a statement using `RepositoryConnection.addStatement()` and the transaction is committed. The program then fetches some statements from the repository using a direct call to the `RepositoryConnection.getStatements()`. The set of retrieved statements should contain the newly added statement since it matches the given pattern. The statement is then removed in a separate transaction.

The application can also be run against a remote repository exposed using the Sesame servlet container. In this case, the `url` parameter is used specify the sesame endpoint and this causes the `config` and `repository` parameters to be ignored. In order to use this, the following extra Apache and Sesame jar files must be included in the classpath (not provided with the OWLIM distribution):

```
commons-logging-1.1.1.jar
commons-codec-1.3.jar
commons-httpclient-3.1.jar
sesame-http-client-2.5.0.jar
```

There are Windows and Linux scripts (namely `example.cmd` and `example.sh`) prepared for convenience that execute the sample application using its default configuration and values.

Wordnet example

Wordnet, is the most popular lexical knowledge base, developed at the University of Princeton. It encodes the meanings of about 150,000 English words. The meanings of the words are defined by word-senses, which relate a word to a lexical concept. Lexical concepts are called *synsets*, i.e. synonym sets – about 115,000 of those appear in Wordnet v.2.0. Numerous lexical semantic relations are formally modelled, e.g.

- Hyponymy (subsumption from a more-general term)
- Antonymy (negation, a term with the opposite meaning)
- Causation and entailment (for verbs)

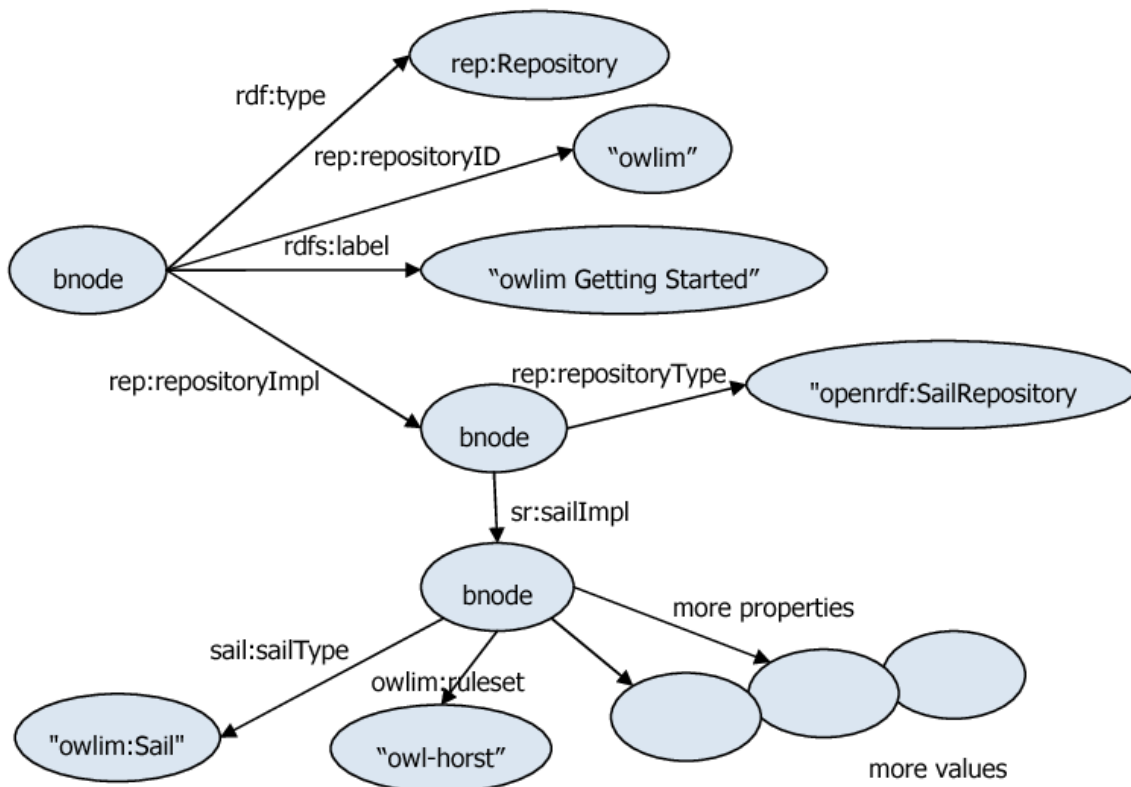
A standard RDF/OWL representation of Wordnet is available at <http://www.w3.org/TR/wordnet-rdf/>. It contains about 1.9 million explicit statements (the Full variant), expressed in a fragment of OWL-Lite that further entails 6.3 million implicit statements.

To configure the getting started example program to use the Wordnet data sets and run the included Wordnet queries, one should download the archive of the full version from <http://www.w3.org/2006/03/wn/wn20/download/wn20full.zip>, extract it into a folder, e.g. `./preload/wordnet` and provide a path to this folder using the `preload` command line parameter when starting the program. Some sample Wordnet queries are provided in the `wordnet.serql` and `wordnet.sparql` query files. These can be specified on the command line using the `queryfile` command line parameter.

Configuration

Sesame 2.0 keeps repository configurations in a SYSTEM repository – in RDF. A new repository can be configured simply by inserting an appropriate graph in to the SYSTEM repository. The getting started application uses the Turtle format for convenience and also because the Sesame console application uses the Turtle format for template files when creating repositories.

The diagram below gives a graphical illustration of an RDF graph that describes a repository configuration:



Often it is desirable to ensure that a repository starts with a predefined set of RDF statements, usually one or more schema graphs. This is possible by using the `owlim:imports` property. After start up, these files are parsed and their contents are permanently added to the repository. The complete set of configuration parameters, their descriptions and their default and allowed values are listed below. What follows is a short description of those properties specific to OWLIM-SE that are used to setup the repository. For more information about Sesame 2 configuration schema refer to the Sesame documentation [9]. In short, the configuration is an RDF graph, the root node is of

`rdf:type rep:Repository`, it must be connected through `rep:RepositoryID` property to a Literal that contains the human readable name of the repository. The root node must be connected via the `rep:repositoryImpl` property to a node that describes the configuration. The type of the repository is defined via `rep:repositoryType` property and its value must be `openrdf:SailRepository` to allow for custom sail implementations (such as OWLIM-SE) to be used in Sesame 2.0. Then a node that specifies the Sail implementation to be instantiated must be connected with `sr:sailImpl` property. To instantiate OWLIM-SE, this last node must have a property `sail:sailType` with the value `owlim:Sail` – the Sesame framework will locate the correct `SailFactory` within the application classpath that will be used to instantiate the Java implementation class.

Namespaces corresponding to the prefixes used in the above paragraph are:

```
rep: <http://www.openrdf.org/config/repository#>
sr: <http://www.openrdf.org/config/repository/sail#>
sail: <http://www.openrdf.org/config/sail#>
owlim: <http://www.ontotext.com/tree/owlim#>
```

All properties used to specify OWLIM-SE's configuration parameters use the `owlim:` prefix and the local names match up with the parameters listed below, e.g. the value of the `ruleset` parameter can be specified using the <http://www.ontotext.com/tree/owlim#ruleset> property.

Many of the OWLIM specific configuration parameters can be set via the Java Virtual Machine (JVM) system properties passed as command line parameters when starting the JVM. Values for configuration parameters that are given on the command line take precedence over those present in the repository configuration. For instance, the `ruleset` parameter can be set from the command line by using:

```
-Druleset=owl-max
```

Sample Configuration

There follows an example configuration (in Turtle RDF format) of a Sesame 2 repository that uses an OWLIM-SE sail implementation:

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

[] a rep:Repository ;
  rep:repositoryID "owlim" ;
  rdfs:label "OWLIM Getting Started" ;
  rep:repositoryImpl [
    rep:repositoryType "openrdf:SailRepository" ;
    sr:sailImpl [
      sail:sailType "owlim:Sail" ;
      owl:ruleset "owl-horst-optimized" ;
      owl:base-URL "http://example.org/owlim#" ;
      owl:imports "./ontology/my_ontology.rdf" ;
      owl:defaultNS "http://www.my-organisation.org/ontology#" ;
      owl:entity-index-size "5000000" ;
      owl:cache-memory "4G" ;
      owl:storage-folder "storage" ;
      owl:repository-type "file-repository" ;
    ]
  ] .
```

Memory Requirements

Apart from the I/O buffers used for caching, OWLIM-SE keeps in memory the indexes from the nodes in the RDF graph. This is a design decision in order to improve the overall performance of the repository. Each I/O buffer (page) is exactly 64kb and the indexing information per node in the graph is 12 bytes. So, depending on the dataset, memory requirements per repository may vary. To ease the calculation for the amount of Java heap memory required for an OWLIM-SE repository an excel spreadsheet is included in the distribution – `owlim-se-configurator.xls`.

The page cache is organized in two sets of buffers, read-only and dirty. Each page is first loaded in to the read-only cache. When this gets full, a page (if dirty) is moved to the dirty cache, where it can be later written to storage.

Cache Memory Configuration

There are several components in OWLIM that make use of caching (e.g. FTS indices, predicate list, tuple indices). In different situations certain caches will need more memory than others. OWLIM allows for the configuration of both the total cache memory to be used by a repository and all the separate per-module caches.

Parameters

The following parameters control the amount of memory assigned to each of the different caches:

Parameter	Unit	Default	Deprecated Equivalent	Description
cache-memory	bytes	none		The amount of memory to be distributed among different caches
tuple-index-memory	bytes	80M	cache-size	Memory used for PSO and POS caches
predicate-memory	bytes	80M		Memory used for predicate list cache
fts-memory	bytes	20M	tokenIndexCacheSize	Memory used for full-text index cache (node search)

All parameters can be specified in bytes, kilobytes, megabytes or gigabytes by using a unit specifier at the end of the integer number. When no unit specifier is given, this is interpreted as bytes, otherwise use k or K - kilobytes, m or M - megabytes and g or G - gigabytes (everything base 2).

Memory Distribution

The general rule of thumb is:

$$\text{cache-memory} = \text{tuple-index-memory} + \text{predicate-memory} + \text{fts-memory}$$

However, if some of the modules using the cache (e.g. full text search) are turned off it is excluded from the above equation. Furthermore, if cache-memory is explicitly configured and some of the other memory parameters are omitted, the missing values are resolved by uniformly distributing the remaining memory after all the explicitly configured memory parameters are subtracted. For example if cache-memory = 100M, fts-memory = 10M and the other memory parameters are missing, then they are implicitly assigned $(100M - 10M) / 2 = 45M$ each. If cache-memory wasn't specified then all the missing memory parameters are assigned their default values.

Backward Compatibility

The old-style parameters that were used to configure cache memory in terms of number of pages are still accepted, although deprecated. When both old and new parameters are used together, the value of the new parameter overrides the value of the old one.

Configuration Parameters

Almost all OWLIM parameters can be set both in the TTL configuration file and from the command line using the Java `-D<param.name>=<value>` command line option to set system properties. When a parameter is set simultaneously using both methods, the system property overrides the value in the configuration file. Some OWLIM parameters can only be set using system properties.

These list of all OWLIM parameters is here:

Parameter	TTL	Java -D	Description
base-URL	X	X	<i>default <none></i> , specifies the default namespace for the main persistence file. Non-empty namespaces are recommended, because their use guarantees the uniqueness of the anonymous nodes that may appear within the repository.
build-pcsot	X	X	<i>default false</i> , by default OWLIM will not build <i>PSOCT</i> and <i>POSCT</i> indices (pred-subj-obj-context-tripletset and pred-obj-subj-context-tripletset); if set to true , this parameter specifies that the <i>PCSOT</i> index will be built, which will speed up extracting all statements from a context.
build-ptsoc	X	X	<i>default false</i> , similar to build-pcsot , but speeds up retrieving all statements from a tripletset.
cache-memory	X	X	<i>default <none></i> , specifies the total amount of memory to be given to all types of cache.
cache-size (DEPRECATED)	X	X	<i>default 4000</i> , works with file repositories only and defines the number of 20k pages in the cache of each of its sorted collections; because there are at least 2 collections (<i>PSO</i> and <i>POS</i> , but <i>PCSOT</i> and <i>PTSOT</i> may be defined as well), the total number of pages is twice (or three times or 4 times) as big.
check-for-inconsistencies	X	X	<i>default false</i> , turns on or off the mechanism for consistency checking; consistency checks are defined in the rule file and are applied at the end of every transaction if this parameter is true .

database-recovery-policy	X	X	<p><i>default</i> recover, specifies the recovery strategy when the repository is initialised and it is detected that the previous instance did not shutdown cleanly. The possible values are:</p> <ul style="list-style-type: none"> • stop – log a message and stop. In this situation a repository must be restored from a backup before it can be re-run; • run – log a message, but continue initialisation. The repository can be used, but some data may have been lost; • recover – log a message and automatically run the database restorer tool to recover data, then continue initialisation.
debug.level		X	<p><i>default</i> 0, defines the level of detail of OWLIM output used in QueryModelConverter, SailConnectionImpl and HashEntityPool as follows:</p> <ul style="list-style-type: none"> • QueryModelConverter: <ul style="list-style-type: none"> • debug.level > 2 : Outputs the query optimization time. • debug.level > 3 : Outputs the query plan. • SailConnectionImpl: <ul style="list-style-type: none"> • debug.level > 0 : Outputs "Owl原因 evaluation strategy" or "Sesame evaluation strategy" when evaluating a query. • debug.level > 2 : ThreadPool outputs when a worker thread starts and stops. • HashEntityPool: <ul style="list-style-type: none"> • debug.level > 1 : If version number is less than the current one, outputs "Older Entity storage version found: X, recent one is: Y".
defaultNS	X		<p><i>default</i> <empty>, default namespaces corresponding to each imported schema file separated by semicolon and the number of namespaces must be equal to the number of schema files from the imports parameter. Example:</p> <p>*owlim:defaultNS "http://www.w3.org/2002/07/owl#;http://example.org/owlim#";*</p> <p>Note: This parameter cannot be set via a command line argument</p>
disable-sameAs	X	X	<i>default</i> false , enables or disables the owl:sameAs optimisation
enable-query-cache (NOT USED)	X	X	<i>default</i> true , enables or disables the query cache, from which the most recent similar query (to the one to be executed) is retrieved to speed up query compilation.
enable-optimization	X	X	<i>default</i> true , enables or disables query optimization.
enablePredicateList		X	<i>default</i> false : enables or disables mappings from an entity (subject or object) to its predicates; switching this on can drastically speed up queries that use wildcard predicate patterns.
enableSmoothDelete	X	X	<i>default</i> true : enables or disables the smooth delete behaviour, see section 7.1.4.
entity-index-size	X	X	<i>default</i> 1000000 , defines the number of entity hash table index entries; the bigger the size, the less the collisions in the hash table and the faster the entity retrieval; the entity hash table does not rehash so its index size is constant throughout the life of the repository.
entity-id-size	X	X	<i>default</i> 32 , possible values are 32 and 40 ; defines the bit size of internal IDs used to index entities (URIs, blank nodes and literals). For most cases, this parameter can be left to its default value. However, if very large datasets are used that contain more than 2^{32} entities, then this parameter should be set to 40 . Be aware, that this can only be set when instantiating a new repository and converting an existing repository between 32 and 40-bit entity widths is not possible.
fts-memory	X	X	<i>default</i> 20m , specifies the amount of memory to be used for the FTS index cache.
ftsIndexPolicy	X	X	<i>default</i> never , possible values are onCommit , onStartup , onShutdown , never ; turns on and off the default mechanism for full text search used via the built-in predicates described in section 10.1; if FTS is turned on then, depending on the value, it determines when the indexing will take place: onCommit - at the end of each transaction, onStartup - at initialisation, onShutdown - on repository shutdown.
ftsLiteralsOnly	X	X	<i>default</i> false , if the Node search (full-text search) mechanism is enabled, this parameter specifies whether only literals will be indexed (value of true , enough in 90% of the cases) or everything (value of false).
hash-table-size (NOT USED)	X	X	<i>default</i> 10000000 , in former times this used to limit the size of the hash table for entities, but now the hash table can grow without limitations.

imports	X		<p>Default none, a list of schema files that will be imported at start up. All the statements, found in these files, will be loaded into the repository and will be treated as read-only. The serialization format is assumed to be RDF/XML, unless the file has a .NT extension. Example:</p> <p>owlim:imports "./ont/owl.rdfs;./ont/ex.rdfs"</p> <p>Schema files can be either a local path name, e.g. ./ontology/myfile.rdf or a URL, e.g. *http://www.w3.org/2002/07/owl.rdf* If this parameter is used then the default namespace for each imported schema file <i>must</i> be provided using the defaultNS parameter. Note: This parameter cannot be set via a command line argument.</p>
in-memory-literal-properties	X	X	<p>default false, turns on and off caching of the literal languages and data-types; if the caching is on and the entity pool is restored from persistence, but there is no such a cache available on disk, it is created after the entity pool initialization.</p>
inferencer-threads	X	X	<p>default 1, if 1 then the default inferencers are used (generated by the RuleCompiler) otherwise multithreaded inferencers are used (generated by the RuleCompilerMT); the number corresponds to the number of threads used for inference.</p>
journaling	X	X	<p>default true, turns on and off transaction persistence on disk; from the persisted transaction the repository can be restored after an unexpected shutdown.</p>
owlim-license	X	X	<p>default <none>, specifies the license file for both OWLIM-SE and OWLIM-Enterprise worker nodes.</p>
partialRDFS (DEPRECATED)	X	X	<p>default false, if true then trivial RDFS inferences are switched off, such as 'every URI is type ' or 'every class is a sub-class of rdfs:Resource', etc. Deprecated in favour of explicit choice of rule-set, see ruleset below. However, if this flag is set to true then the chosen ruleset will effectively have -optimized appended to it if not present, e.g. ruleset=owl-horst and partialRDFS=true is the same as selecting ruleset=owl-horst-optimized</p>
predicate-list-cache-size (DEPRECATED)	X	X	<p>default 1000, defines the number of 20k pages used for caching the predicate lists (these are lists of predicates per entity used as subject or object, i.e. subjects and objects are mapped to a list of predicates; this can speed up queries that use wildcard predicate patterns).</p>
predicate-memory	X	X	<p>default 80m, specifies the amount of memory to be used for predicate lists cache.</p>
query-timeout	X	X	<p>default -1 (infinity), sets the number of seconds after which the evaluation of a query will be terminated; negative values stand for infinity.</p>
rebuild.rules		X	<p>default true, where true instructs the rule compiler to write the generated inference classes to the src directory of the Java project (if one wants to start the RuleCompiler then he/she must use this parameter with value of true); this parameter is false when used by the RuntimeInferencerCompiler; used in RuleCompiler and RuleCompilerMT.</p>
repository-fragments	X	X	<p>default 1, works with a file repository only. If it is set to 1 then the default AVLRepository will be used, otherwise a fragmented repository will be used; this can speed up statement storage and retrieval. This parameter controls the number of pieces that the repository indexes are sliced in to, where each slice is stored in a separate folder.</p>
repository-type	X	X	<p>default file-repository, available values are: file-repository weighted-file-repository</p>
ruleset	X	X	<p>default owl-horst, built-in rule sets are empty, rdfs, owl-horst, owl-max and owl2-rl and their optimized counterparts rdfs-optimized, owl-horst-optimized, owl-max-optimized and owl2-rl-optimized. A custom rule set is chosen by setting the path to its rule file (.pie).</p>
storage-folder	X	X	<p>default none, specifies the folder where the index files will be stored.</p>
tokenIndexCacheSize (DEPRECATED)	X	X	<p>default 1000, defines the number of pages to be used by the default full text search; this is now deprecated and replaced by fts-memory which specifies the memory available for FTS, so the number of pages is automatically calculated.</p>
tokenization-regex	X	X	<p>default [p{L}d_]+, defines the rule for splitting strings into tokens; the tokens themselves, not the strings, are stored in the FTS index, so it is important to define a suitable tokenization for the given application (e.g. in some cases intensive number parsing is used, in other cases - personal names, i.e. ones starting with capital letter and may contain hyphens and apostrophes, etc.)</p>
tuple-index-memory	X	X	<p>default 80m, specifies the amount of memory to be used for statement storage cache.</p>

useShutdownHooks	X	X	<i>default true</i> , if <i>true</i> then the method OwlImSchemaRepository.shutdown() is called when the Java Virtual Machine (JVM) exits (running OWLIM under Tomcat requires this parameter to be true , otherwise it cannot be guaranteed that the shutdown() method will be called at all).
-------------------------	---	---	--

Performance analytics

OWLIM-SE can provide information about its internal state and behaviour, including trend information. Such information can be useful for 'debugging' certain situations, such as understanding why load performance changes over time or with particular data sets.

Statistics are kept for the main index data structures and includes information such as: cache hits/misses, file reads/writes, etc. This information can be used to fine tune OWLIM-SE's memory configuration.

The analytical information is published via a Java management extensions **JMX** management bean (MBean). A JMX endpoint is configured by using special system properties when starting the Java virtual machine (JVM) in which OWLIM-SE is running. For example, the following command line parameters will set the JMX server endpoint to listen on port 8089, not require authentication and will not use a secure socket layer:

```
-Dcom.sun.management.jmxremote.port=8089
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

If using OWLIM-SE with Tomcat, then these parameters must be passed to Tomcat's JVM by setting either the `JAVA_OPTS` or `CATALINA_OPTS` environment variable, e.g.

```
set JAVA_OPTS="-Dcom.sun.management.jmxremote.port=8089
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
```

Once OWLIM-SE is loaded, use any compliant JMX client, e.g. `jconsole` that is part of the Java development kit, to access the JMX interface on the configured port. The entry point for accessing analytical information will be advertised via the `com.ontotext.AVLRepositoryManager` MBean, which has a single attribute called 'CollectionsStatisticsFlag'. By default, this flag is set to `false`. After setting this value to `true`, statistics will be maintained for all the major indices. For each index, there will be a `SortedCollectionStatistics` MBean published that shows the cache and file I/O values updated in real-time.

The following information is displayed for each index:

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

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

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

OWLIM-SE Administrative Tasks

This section contains information about performing day to day administration tasks with an OWLIM-SE repository. Most of these standard operations can be achieved using the Sesame software. Some information is repeated here and tailored to the specifics of OWLIM-SE.

- [Preserving Context when Exporting and Importing Data](#)
- [Loading large amounts of data](#)
- [Dumping a large repository to RDF](#)
- [Migration of a data from incompatible versions](#)
- [Modifying a repository's configuration after it has been created](#)
- [Restoring the repository after a crash](#)
- [Flush to disk without shutting down the repository](#)
- [Exception thrown during query answering or initialisation](#)
- [How can the rule set be changed](#)
- [Backing up and restoring the repository](#)
- [Modifying an imported ontology/schema](#)

Preserving Context when Exporting and Importing Data

In order to preserve the context across export/import a context-aware RDF file format must be used, e.g. TriG. After serialising the repository to a file with this format (this can be done through the Sesame workbench Web application) the file can be imported with the following steps:

- Go to [Add]
- Choose Data format: TriG
- Choose RDF Data File: e.g. export.trig
- Clear the context text field (it will have been set to the URL of the file). If this is not cleared then all the imported RDF statements will be given a context of `<file://export.trig>` or similar.
- Upload

The TriX format (an XML-based context-aware RDF serialisation) can also be used.

Loading large amounts of data

In general RDF data can be loaded into a given Sesame repository using the 'load' command in the Sesame console application or directly through the workbench web application. However, neither of these approaches will work when using a very large number of triples, e.g. a billion statements. A common solution would be to convert the RDF data into a line-based RDF format (e.g. N-triples) and then split it into many smaller files (e.g. using the linux command 'split'). This would allow each file to be uploaded separately using either the console or workbench applications.

Dumping a large repository to RDF

The Sesame openRDF workbench Web application has an export function that can be used to export the contents of moderately sized repositories. However, using this with large repositories (more than a hundred million statements or more) causes problems, usually timeouts for the Servlet container (Tomcat) hosting the application. Also, the workbench cannot be used when using OWLIM-SE without Tomcat.

Therefore, a more straightforward approach for exporting RDF data from repositories is to do this programmatically. The Sesame `RepositoryConnection.getStatements()` method can be called with the `includeInferred` flag set to `false` (in order not to serialise the inferred statements). Then the returned iterator can be used to visit every explicit statement in the repository and one of the Sesame RDF writer implementations can be used to output the statements in the chosen format. If the data will be re-imported, the N-Triples format is recommended, because this can easily be broken in to large 'chunks' that can be inserted and committed separately. The following code snippet shows how an export can be achieved using this approach:

```
java.io.OutputStream out = ...;
RDFWriter writer = Rio.createWriter(RDFFormat.NTRIPLES, out);
writer.startRDF();
RepositoryResult<Statement> statements =
repositoryConnection.getStatements(null, null, null, false);
while (statements.hasNext()) {
    writer.handleStatement(statements.next());
}
statements.close();
writer.endRDF();
out.flush();
```

Migration of a data from incompatible versions

The basic procedure is to export the RDF data from the old version of OWLIM-SE and then reload it in to a new repository instance that uses the new version of OWLIM-SE. Exporting is straightforward when using the Sesame workbench – simply click the 'Export' button, choose the format and click 'download'. To import in to a new repository, click 'add', select a format, specify the file and base URI, then click 'Upload'.

If not using the Sesame workbench, the export must be done programmatically using the

`RepositoryConnection.getStatements()` API, because the Sesame console does not have an export function. NOTE: It should be possible to export only the explicit statements, as the inferred statements will be recomputed at load time. Fortunately, the Sesame console application does have a 'load' function and this can be used to reload the exported statements.

Modifying a repository's configuration after it has been created

Once created, the repository configuration is maintained in the Sesame SYSTEM repository. There is no easy generic way of changing this configuration, but there are several possibilities.

Firstly, OWLIM-SE allows most of the configuration parameters to be overridden at runtime by specifying the parameter values as JVM options. For example, to change the cache-memory configuration parameter, pass `-Dcache-memory=1g` option to the JVM that is hosting the application using OWLIM-SE.

The second approach is to modify the SYSTEM repository directly. Caution, make sure that the SYSTEM repository is not corrupted and that no other repository configurations are damaged. There are no tools for doing this, but it is probably easiest (if the current configuration is known) to just remove the repository configuration using the Sesame console (drop `<repository_id>`) or the Sesame workbench WebApp. Then the configuration in the TTL file can be modified and added to the SYSTEM repository (keeping the same repository id).

Restoring the repository after a crash

A OWLIM-SE repository image can become corrupted if OWLIM-SE (or the Java application that hosts it) crashes. In particular, the repository gets corrupted if OWLIM-SE is interrupted after a successful commit but before flushing the update to disk.

By default, if OWLIM-SE detects that the last shutdown was not normal, it will attempt to restore its internal state back to the end of the last successfully committed transaction. Therefore, users will not normally need to invoke a restore process manually. However, if the `database-recovery-policy` parameter has been changed to anything other than `recover`, then no attempt will be made to restore any lost data. In this situation, the user may want to invoke a restore process manually as follows.

If an abnormal termination occurs, it is advisable to execute a database restore as soon as possible. This can also help if OWLIM-SE fails to load a repository during startup for some reason. The OWLIM-SE jar contains a utility for restoring database images. It can be executed in the root folder of the OWLIM-SE distribution as follows:

```
java -cp lib/*:ext/* com.ontotext.trree.DatabaseRestorer STORAGE_FOLDER_PATH ENTITY_INDEX_SIZE
RULE_SET
```

The following three parameters are required:

STORAGE_FOLDER_PATH	The full path to the repository storage directory.
ENTITY_INDEX_SIZE	The value of the entity-index-size parameter in the TTL repository configuration file.
RULE_SET	The value of the ruleset parameter in the TTL repository configuration file.

Flush to disk without shutting down the repository

To ensure that all committed statements are written to disk storage, a transaction can be committed containing a statement with the special predicate `<http://www.ontotext.com/owlim/system#flush>`, e.g.

```
<http://www.example.com> <http://www.ontotext.com/owlim/system#flush> ""
```

This statement can be committed with other statements as a single transaction or separately in its own transaction. All repository data is flushed to disk after the transaction is committed.

Exception thrown during query answering or initialisation

If the Lucene jar file is not on the classpath then the following exception will be thrown:

```
java.lang.NoSuchMethodError: org.apache.lucene.queryParser.QueryParser
The Lucene.jar file must be added to the Java classpath, even when not using full-text search.
```

How can the rule set be changed

Changing the rule-set can be achieved in the same way as changing any other configuration parameter, except that it is necessary to re-compute the inferred statements with the new rule-set. This does not happen automatically, but can be forced by committing a transaction containing a statement with the special re-infer predicate (the subject and object can be anything), e.g.

```
<http://www.example.com> <http://www.ontotext.com/owlim/system#reinfer> ""
```

Backing up and restoring the repository

There is no facility at present to back up a repository while it is running. Therefore, to backup a repository it must be shutdown gracefully and a copy of its storage directory (and any sub-directories) taken.

If the Sesame `RepositoryManager` class is being used, the repository directory will usually be in a directory path that ends with 'repositories/<repository_id>/owlim-storage', but this can be overridden with the `storage-folder` configuration parameter.

To restore a repository from a back up, make sure the repository is not running and then replace the entire contents of the storage directory (and any sub-directories) with the backup. Then restart the repository and check the log file to ensure a successful start up.

Modifying an imported ontology/schema

Ontologies and schemas imported at initialisation time using the 'imports' configuration parameter are flagged as read-only. This is the preferred method for using ontologies/schemas as it allows the incremental delete algorithm to be very efficient. However, there are times when it is necessary to change a schema and this can be done inside a 'system transaction'.

The user instructs OWLIM that the transaction is a system transaction by including a dummy statement with the special schema-transaction predicate, i.e.

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

This statement is not inserted in to the database, rather it serves as a flag that tells OWLIM that it can ignore the read-only flag for imported statements.

OWLIM-SE Access Rights and Security

- [Operations](#)
- [Security constraints and roles](#)
- [User accounts](#)
- [Programmatic authentication](#)

Controlling access to an OWLIM repository and assigning user accounts to roles with specified access permissions is achieved with a combination of HTTP authentication and the Sesame server's deployment descriptor. Sesame and OWLIM have no separate access control mechanisms, therefore to control access, OWLIM must be deployed using the Sesame HTTP server.

Using this technique will allow an administrator to:

- Create security constraints on operations (reading statements, writing statements, modifying named graphs, changing namespace definitions, etc)
- Group security constraints in to security roles
- Manage user accounts and assign these to specific security roles

Operations

The Sesame HTTP server is a servlet-based Web application deployed to any standard servlet container - for the remainder of this section it is assumed that Tomcat is being used.

The Sesame server exposes its functionality using a [RESTful](#) API that is an extension of the [SPARQL protocol for RDF](#). This protocol defines exactly what operations can be achieved using specific URL patterns and HTTP methods (GET, POST, PUT, DELETE). Each combination of URL pattern and HTTP method can be associated with a set of user roles, thus giving very fine-grained control.

The following diagram shows the URL patterns associated with groups of operations on a Sesame HTTP server.

```
<SESAME_URL>
  /protocol           : protocol version (GET)
  /repositories       : overview of available repositories (GET)
  /<REP_ID>           : query evaluation on a repository (GET/POST)
    /statements       : repository statements (GET/POST/PUT/DELETE)
    /contexts         : context overview (GET)
    /size              : #statements in repository (GET)
    /namespaces       : overview of namespace definitions (GET/DELETE)
      /<PREFIX>       : namespace-prefix definition (GET/PUT/DELETE)
```

This diagram is copied from chapter 8 of the [online Sesame system guide, chapter 8](#).

In general, read operations are effected using GET and with PUT, POST and DELETE for write operations. The exception to this is that POST is allowed for SPARQL queries. This is for practical reasons, because some HTTP servers have limits on the length of parameter values for GET requests.

Security constraints and roles

The association between operations and security roles is specified using security constraints in the Sesame server's deployment descriptor - a file called `web.xml` that can be found in the `.../webapps/openrdf-sesame/WEB-INF` directory.



`web.xml` will be present immediately after installation without any security roles defined. If this file is edited, then care must be taken to create a backup - if the Sesame HTTP server is redeployed then it will over-write `web.xml` with the default version. In particular, do not edit this file while Tomcat is running.

The deployment descriptor defines:

- authentication mechanism/configuration
- security constraints in terms of operations (URL pattern plus HTTP method)
- security roles associated with security constraints

To enable authentication, add the following XML element to `web.xml` inside the `<web-app>` element:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Sesame</realm-name>
</login-config>
```

Security constraints associate operations (URL pattern plus HTTP method) with security roles. Both security constraints and security roles are nested in the `<web-app>` element.

A security constraint minimally consists of a collection of web resources (defined in terms of URL patterns and HTTP methods) and an authorisation constraint (the role name that has access to the resource collection). Some example security constraints are shown below:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SPARQL query access to the 'test' repository</web-resource-name>
    <url-pattern>/repositories/test</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>viewer</role-name>
    <role-name>editor</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      Read access to 'test' repository's namespaces, size, contexts, etc
    </web-resource-name>
    <url-pattern>/repositories/test/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>viewer</role-name>
    <role-name>editor</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Write access</web-resource-name>
    <url-pattern>/repositories/test/*</url-pattern>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>editor</role-name>
  </auth-constraint>
</security-constraint>
```

The ability to create and delete repositories requires access to the SYSTEM repository. An administrator security constraint for this would look like the following:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Administrator access to SYSTEM</web-resource-name>
    <url-pattern>/repositories/SYSTEM</url-pattern>
    <url-pattern>/repositories/SYSTEM/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
```

Also nested inside the `<web-app>` element are definitions of security roles. The format is best shown by example:

```
<security-role>
  <description>
    Read only access to repository data
  </description>
  <role-name>viewer</role-name>
</security-role>

<security-role>
  <description>
    Read/write access to repository data
  </description>
  <role-name>editor</role-name>
</security-role>

<security-role>
  <description>
    Full control over the repository, as well as creating/deleting repositories
  </description>
  <role-name>administrator</role-name>
</security-role>
```

User accounts

Tomcat has a number of ways to manage user accounts. The different techniques are called 'realms' and the default one is called 'UserDatabaseRealm'. This is the simplest one to manage, but also the least secure, because usernames and passwords are stored in plain text.

For the default security realm, usernames and passwords are stored in the file `tomcat-users.xml` in the Tomcat configuration directory, usually `/etc/tomcat6/tomcat-users.xml` on Linux systems. To add user accounts, `<user>` elements must be added inside the `<tomcat-users>` element, for example:

```
<user username="adam" password="secret" roles="viewer" />
<user username="eve" password="password" roles="viewer,editor,administrator" />
```

Programmatic authentication

To use a remote repository where authentication has been enabled, it is necessary to provide the username and password to the Sesame API. Remote repositories will usually be accessed via the `org.openrdf.repository.manager.RemoteRepositoryManager` class. Before attempting access, it is necessary to tell the repository manager the security credentials using the following method:

```
void setUsernameAndPassword(String username, String password)
```

Alternatively, they can be passed in the factory method:

```
static RemoteRepositoryManager getInstance(String serverURL, String username, String password)
```

OWLIM-SE Full-text Search

- [Node Search - Proprietary Full-Text Search](#)
- [RDF Search - Full-Text Search using Lucene](#)

Full-text search (FTS) concerns retrieving text documents out of a large collection by keywords or, more generally, by tokens (represented as sequences of characters). Formally, the query represents an unordered set of tokens and the result is set of documents, relevant to the query. In a simple FTS implementation, relevance is Boolean: a document is either relevant to the query, when it contains all the query tokens, or not. More advanced FTS implementations deal with a degree of relevance of the document to the query, usually judged on some sort of measure of the frequency of appearance of each of the tokens in the document, normalized versus the frequency of their appearance in the entire document collection. Such implementations return an ordered list of documents, where the most relevant documents come first.

FTS and structured queries, like those in database management systems (DBMS), are different information access methods based on a different query syntax and semantics, where the results are also displayed in a different form. FTS and databases usually require different types of indices too. The ability to combine these two types of information access methods is very useful for a wide range of applications. Many relational DBMS support some sort of FTS (which is integrated into the SQL syntax) and maintain additional indices that allow efficient evaluation of FTS constraints. Typically, relational DBMS allow the user to define a query, which requires specific tokens to appear in a specific column of a specific table. In SPARQL there is no standard way for the specification of FTS constraints. In general, there is neither a well defined nor widely accepted concept for FTS in RDF data. Nevertheless, some semantic repository vendors offer some sort of FTS in their engines. This section documents the FTS supported by OWLIM-SE.

Two approaches are implemented in OWLIM-SE, a proprietary implementation called 'Node Search', and a Lucene-based implementation called 'RDF Search'. The two approaches are collectively referred to in this guide as 'full-text indexing' and both of them enable OWLIM to perform complex queries against character data, which significantly speeds up the query process. To select one of them, one should consider their functional differences, which are outlined in the table below. Furthermore, there can be considerable differences between indexing and search speed of the two FTS implementations. Thus, performance-conscious users are recommended to experiment with the performance of both methods with respect to dataset and queries representative for the intended application.

	Node Search	RDF Search
FTS query form	List of tokens	List of tokens (with Lucene query extensions)
Result form	Unordered set of nodes	Ordered list of URIs
Textual Representation	For literals: the string value. For URIs and B-nodes: tokenized URL	Concatenation of the text representations of the nodes from the molecule (1-step neighbourhood in the graph) of the URI
Relevance	Boolean, based on presence of the query tokens in the text	Vector-space model, reflecting the degree of relevance of the text and the RDF rank of the URI
Implementation	Proprietary full-text indexing and search implementation	The Lucene engine is integrated and used for indexing and search

The Node Search (with parameter **ftsLiteralsOnly** set to **true**) resembles functionality similar to typical FTS implementations in relational DBMS. However, RDF Search is a novel information retrieval concept, which allows for efficient extraction of RDF resources from huge datasets, where ordering of the results by relevance is crucial.

Node Search – Proprietary Full-Text Search

The parameters for OWLIM's full-text index control when/if the index is to be created, the index cache size, and whether literals only or all types of nodes should be indexed. See the parameters **ftsIndexPolicy**, **fts-memory** and **ftsLiteralsOnly** in the [configuration section](#). The following example configures the database engine to create a 20 megabyte cache for the full-text index on start up that indexes all literals and URIs:

```
owlim:ftsIndexPolicy "onStartup" ;
owlim:fts-memory "20m" ;
owlim:ftsLiteralsOnly "false"
```

Full-text search patterns are embedded in SPARQL and SeRQL queries by adding extra statement patterns that use special system predicates:

```
<String:> <Algorithm predicate> <Binding> .
```

Each of the elements of this triple is explained below:

- **<String:>** the search string - a list of tokens separated by colons ':', whose use is determined by the choice of predicate, see below;

- `<Algorithm predicate>` specifies the search method, i.e. how the tokens in the search string are to be used, see below;
- `<Binding>` the variable containing the result, i.e. the values (URIs or literals) that match with the given search string and method.

Predicate	Description
<code>fts:exactMatch</code>	Matches literals that contain all tokens considering the case. For example, searching for <code><United:States></code> will match "The president of the United States", but not "United Statesless", "united states" or "notUnited notStates".
<code>fts:matchIgnoreCase</code>	Similar to the above but ignores case. <code><United:States></code> will match "The president of the United States", "united states" but not "United Statesless" or "notUnited notStates".
<code>fts:prefixMatch</code>	Matches tokens that begin with the given search tokens considering the case. For example, <code><United:States></code> will match "The president of the United States" and "United Statesless" but not "notUnited notStates" or "united states".
<code>fts:prefixMatchIgnoreCase</code>	Similar to the above but ignores case. For example, <code><United:States></code> will match "The president of the United States", "United Statesless", "united states" but not "notUnited notStates".

The namespace prefix `onto` in the above table `<http://www.ontotext.com/owlim/fts#>`

There follow some query examples for Node search in SPARQL and SeRQL:

- **Example 1:** Get all values that contain a token that matches exactly with 'abstract'
SPARQL query:

```
PREFIX fts: <http://www.ontotext.com/owlim/fts#>
SELECT ?label
WHERE { <abstract:> fts:exactMatch ?label . }
```

SeRQL query:

```
SELECT L
FROM {<abstract:abstract>} fts:exactMatch {L}USING NAMESPACE
fts = <http://www.ontotext.com/owlim/fts#>
```

Note that in SeRQL, **abstract:** is not a valid URI, so **abstract:abstract** is used instead, which works the same and also conforms with what the parser expects.

- **Example 2:** Get all values that contain both tokens 'Remorselessness' and 'books' using case-insensitive search (SPARQL):

```
PREFIX fts: <http://www.ontotext.com/owlim/fts#>
SELECT ?label
WHERE { <Remorselessness:books> fts:matchIgnoreCase ?label. }
```

The corresponding SeRQL query is omitted due to its similarity with the above SPARQL query.

- **Example 3:** Find everything that has a label that starts with "3d" regardless of the language or the case (SPARQL):

```
PREFIX fts: <http://www.ontotext.com/owlim/fts#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label WHERE {
  ?X rdfs:label ?label .
  <3d:> fts:prefixMatchIgnoreCase ?label. }
```

This query cannot be expressed in SeRQL using full-text search predicates, because the SeRQL parser won't accept a URI starting with a digit.

The above example is hard to formulate without a full text search capability. For example, the trivial query below won't match an entry with the label "3d"@en, because this literal is an `rdf:PlainLiteral` and not the same as "3d", which is an `xsd:string`, i.e. the data types are different.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?x
WHERE { ?x rdfs:label "3d" }
```

RDF Search - Full-Text Search using Lucene

Apache Lucene is a high-performance, full-featured text search engine written entirely in Java. OWLIM-SE supports full text search capabilities using Lucene with a variety of indexing options and the ability to simultaneously use multiple, differently configured indices in the same query.



The classpath must include lucene-core-3*.jar (included with the OWLIM distribution) in order for the Lucene-based full-text search to function correctly. This can be substituted with the full Lucene jar file that can be downloaded from the [Apache Lucene download page](#).

In order to use Lucene full-text search in OWLIM-SE a Lucene index must first be computed. Before being created, each index can be parameterised in a number of ways using SPARQL ASK queries. This provides the ability to:

- select what kinds of nodes are indexed (URIs/literals/blank-nodes)
- select what is included in the 'molecule' (explained below) associated with each node
- select literals with certain language tags
- choose the size of the RDF 'molecule' to index
- choose whether to boost the relevance of nodes using RDF Rank values
- select alternative analysers
- select alternative scorers

In order to use the indexing behaviour of Lucene, a text document must be created for each node in the RDF graph to be indexed. This text document is called the 'RDF molecule' and is made up of other nodes reachable via the predicates that connect nodes to each other. Once a molecule has been created for each node, Lucene creates an index over these molecules. During search (query answering) Lucene identifies the matching molecules and OWLIM uses the associated nodes as variables substitutions when evaluating the enclosing SPARQL query.

The scope of an RDF molecule includes the starting node and its neighbouring nodes that are reachable via the specified number of predicate arcs. What type of nodes are indexed and what type of nodes are included in the molecule can be specified for each Lucene index. Furthermore, the size of the molecule can be controlled by specifying the number of allowed traversals of predicate arcs starting from the molecule centre (the node being indexed). Note that blank nodes themselves are never included in the molecule. If a blank node is encountered the search is extended via any predicate to the next nearest entity and so on. Therefore even when the molecule size is 1, entities reachable via several intermediate predicates can still be included in the molecule if all the intermediate entities are blank nodes. The parameters are described in more detail as follows:

Parameter	Exclude
Predicate	http://www.ontotext.com/owlim/lucene#exclude
Description	Provides a regular expression to identify nodes that will be excluded from the molecule. Note that the regular expression will be applied case-sensitively to literals and URI local names. The example given below will cause matching URIs (e.g. http://example.com/uri#helloWorld) and literals (e.g. "hello world!") not to be included.
Default	<none>
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:exclude luc:setParam "hello.*" }

Parameter	Exclude entities
Predicate	http://www.ontotext.com/owlim/lucene#excludeEntities
Description	A comma/semi-colon/white-space separated list of entities that will NOT be included in an RDF molecule. The example below will include any URI in a molecule, except the two listed.
Default	<none>
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:excludeEntities luc:setParam "http://www.w3.org/2000/01/rdf-schema#Class http://www.example.com/dummy#E1 " " }

Parameter	Exclude predicates
Predicate	http://www.ontotext.com/owlim/lucene#excludePredicates
Description	A comma/semi-colon/white-space separated list of properties that will NOT be traversed in order build an RDF molecule. The example below will prevent any entities being added to an RDF molecule if they can only be reached via the two given properties.

Default	<none>
Example	<pre>PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:excludePredicates luc:setParam "http://www.w3.org/2000/01/rdf-schema#subClassOf http://www.example.com/dummy#p1" }</pre>

Parameter	Include
Predicate	http://www.ontotext.com/owlim/lucene#include
Description	Indicates what kinds of nodes are to be included in the molecule. The value can be a list of values from: uri, literal, centre (the plural forms are also allowed: uris, literals, centres). The value of <i>centre</i> causes the node for which the molecule is built to be added to the molecule (provided it is not a blank node). This can be useful, for example, when indexing URI nodes with molecules that contain only literals, but the local part of the URI should also be searchable.
Default	"literals"
Example	<pre>PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:include luc:setParam "literal uri" . }</pre>

Parameter	Include entities
Predicate	http://www.ontotext.com/owlim/lucene#includeEntities
Description	A comma/semi-colon/white-space separated list of entities that can be included in an RDF molecule. Any other entities will be ignored. The example below will build molecules that only contain the two entities.
Default	<none>
Example	<pre>PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:includeEntities luc:setParam "http://www.w3.org/2000/01/rdf-schema#Class http://www.example.com/dummy#E1" }</pre>

Parameter	Include predicates
Predicate	http://www.ontotext.com/owlim/lucene#includePredicates
Description	A comma/semi-colon/white-space separated list of properties that can be traversed in order build an RDF molecule. The example below will allow any entities to be added to an RDF molecule, but only if they can be reached via the two given properties.
Default	<none>
Example	<pre>PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:includePredicates luc:setParam "http://www.w3.org/2000/01/rdf-schema#subClassOf http://www.example.com/dummy#p1" }</pre>

Parameter	Index
Predicate	http://www.ontotext.com/owlim/lucene#index
Description	Indicates what kinds of nodes are to be indexed. The value can be a list of values from: uri, literal, bnode (the plural forms are also allowed: uris, literals, bnodes).
Default	"literals"
Example	<pre>PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:index luc:setParam "literals, bnodes" . }</pre>

Parameter	Language(s)
Predicate	http://www.ontotext.com/owlim/lucene#languages
Description	A comma separated list of language tags. Only literals with the indicated language tags will be included in the index. To include literals that have no language tag, use the special value 'none'.
Default	"" (which is used to indicate that literals with any language tag are used, including those with no language tag)
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:languages luc:setParam "en,fr,none" . }

Parameter	Molecule size
Predicate	http://www.ontotext.com/owlim/lucene#moleculeSize
Description	Set the size of the molecule associated with each entity. A value of zero indicates that only the entity itself should be indexed. A value of 1 indicates that the molecule will contain all entities reachable by a single 'hop' via any predicate (predicates not included in the molecule). Note that blank nodes themselves are never included in the molecule. If a blank node is encountered the search is extended via any predicate to the next nearest entity and so on. Therefore even when the molecule size is 1, entities reachable via several intermediate predicates can still be included in the molecule if all the intermediate entities are blank nodes. Molecule sizes of 2 and upwards are allowed, but with large datasets it can take a very long time to create the index.
Default	0
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:moleculeSize luc:setParam "1" . }

Parameter	Use RDF rank
Predicate	http://www.ontotext.com/owlim/lucene#useRDFRank
Description	Indicates whether the RDF weights (if they have been computed already) associated with each entity should be used as boosting factors when computing the relevance to a given Lucene query. Allowable values are 'no', 'yes' and 'squared'. This last value indicates that the square of the RDF Rank value is to be used.
Default	"no"
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:useRDFRank luc:setParam "yes" . }

Parameter	Set alternative analyser
Predicate	http://www.ontotext.com/owlim/lucene#analyzer
Description	Used to set an alternative analyser for processing text to produce terms to index. By default, this parameter has no value and the default analyser used is: org.apache.lucene.analysis.standard.StandardAnalyzer An alternative analyser must be derived from: org.apache.lucene.analysis.Analyzer To use an alternative analyser, use this parameter to identify the name of a Java factory class that can instantiate it. The factory class must be available on the Java virtual machine's classpath and must implement this interface: com.ontotext.trree.plugin.lucene.AnalyzerFactory
Default	<none>
Example	PREFIX luc: < http://www.ontotext.com/owlim/lucene# > ASK { luc:analyzer luc:setParam "com.ex.MyAnalyserFactory" . }

Parameter	Set alternative scorer
Predicate	http://www.ontotext.com/owlim/lucene#scorer

Description	Used to set an alternative scorer that provides boosting values that adjust the relevance (and hence the ordering) of results to a Lucene query. By default, this parameter has no value and no additional scoring takes place, however, if the useRDFRank parameter is set to true, then the RDF Rank scores are used (see section 10.1). An alternative scorer must implement this interface: com.ontotext.trree.plugin.Scorer In order to use an alternative scorer, use this parameter to identify the name of a Java factory class that can instantiate it. The factory class must be available on the Java virtual machine's classpath and must implement this interface: com.ontotext.trree.plugin.ScorerFactory
Default	<none>
Example	PREFIX luc: <http://www.ontotext.com/owlim/lucene#> ASK { luc:scorer luc:setParam "com.ex.MxScorerFactory" . }

Once the parameters for an index have been set, the index is created and named using a SPARQL ASK query of this form, where the index name appears as the subject in the query statement pattern:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
ASK { luc:myIndex luc:createIndex "true" . }
```

The index name must have the <http://www.ontotext.com/owlim/lucene#> namespace and the local part can contain only alphanumeric characters and underscores.

Creating an index can take some time, although usually no more than a few minutes when the molecule size is 1 or less. During this process, for each node in the repository its surrounding molecule is computed. Then each such molecule is converted into a single string document (by concatenating the textual representation of all the nodes in the molecule) and this document is indexed by Lucene. If RDF Rank weights are used (or an alternative scorer is specified) then the computed values are stored in Lucene's index as a boosting factor that will later on influence the selection order.

To use a custom Lucene index in a SPARQL query use the index's name as the predicate in a statement pattern, with the Lucene query as the object using the full [Lucene query](#) vocabulary.

The following query will produce bindings for ?s from entities in the repository, where the RDF molecule associated with that entity (for the given index) contains terms that begin with "United". Furthermore, the bindings will be ordered by relevance (with any boosting factor):

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
SELECT ?s
WHERE { ?s luc:myIndex "United*" . }
```

The Lucene score for a bound entity for a particular query can be exposed using a special predicate:

```
http://www.ontotext.com/owlim/lucene#score
```

This can be useful when lucene query results should be ordered in a manner based on, but different from, the original Lucene score. For example, the following query will order results by a combination of the Lucene score and some ontology defined importance value:

```
PREFIX luc: <http://www.ontotext.com/owlim/lucene#>
PREFIX ex: <http://www.example.com/myontology#>
SELECT * {
  ?node luc:myIndex "lucene query string" .
  ?node ex:importance ?importance .
  ?node luc:score ?score .
} ORDER BY ( ?score + ?importance )
```

The `luc:score` predicate will work only on bound variables. There is no problem disambiguating multiple indices because each variable will be bound from exactly one Lucene index and hence its score.

The combination of ranking RDF molecules together with full-text search provides a powerful mechanism for querying/analysing datasets even when the schema is not known. This allows for keyword-based search over both literals and URIs with the results ordered by importance/interconnectedness. For an example of this kind of 'RDF Search', see [FactForge](#).

OWLIM-SE Geo-spatial Extensions

- [Geo-spatial query syntax](#)
- [Extension query functions](#)
- [Implementation details](#)

OWLIM-SE has special support for 2-dimensional geo-spatial data that uses the [WGS84 Geo Positioning RDF vocabulary](#) ([World Geodetic System 1984](#)). Special indices can be used for this data that permit the efficient evaluation of special query forms and extension functions that allow:

- locations to be found that are within a certain distance of a point, i.e. within the specified circle on the surface of the sphere (Earth), using the `nearby(...)` construction;
- locations that are within rectangles and polygons, where the vertices are defined using spherical polar coordinates, using the `within(...)` construction.



The following jar files (included with the OWLIM distribution) must be on the classpath in order for the geo-spatial extensions to function correctly: `jsi-1*.jar`, `log4j-1*.jar`, `sil-0*.jar`, `trove4j-2*.jar`

The WGS84 ontology can be found at: http://www.w3.org/2003/01/geo/wgs84_pos and contains several classes and predicates:

Element	Description
<code>SpatialThing</code>	Class used to represent anything with spatial extent, i.e. size, shape or position.
<code>Point</code>	Class used represent a point (relative to Earth) defined using latitude, longitude (and altitude). subClassOf http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
<code>location</code>	The relation between a thing and where it is. range <code>SpatialThing</code> subPropertyOf http://xmlns.com/foaf/0.1/based_near
<code>lat</code>	The WGS84 latitude of a <code>SpatialThing</code> (decimal degrees). domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
<code>long</code>	The WGS84 longitude of a <code>SpatialThing</code> (decimal degrees). domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
<code>lat_long</code>	A comma-separated representation of a latitude, longitude coordinate.
<code>alt</code>	The WGS84 altitude of a <code>SpatialThing</code> (decimal meters above the local reference ellipsoid). domain http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing

Before the geo-spatial extensions can be used, the geo-spatial index must be built. This is achieved using a special predicate as follows:

```
PREFIX ontogeo: <http://www.ontotext.com/owlim/geo#>
ASK { _:b1 ontogeo:createIndex _:b2. }
```

If the indexing is successful, the above query will return true, false otherwise. Information about the indexing process and any errors can be found in the log. Note that this query will return false if there is no geospatial data in the repository, i.e. no statements describing resources with latitude and longitude properties.

Geo-spatial query syntax

The special syntax used to query geo-spatial data makes use of SPARQL's [RDF Collections syntax](#). This syntax uses round brackets as a shorthand for the statements connecting a list of values using `rdf:first` and `rdf:rest` predicates with terminating `rdf:nil`. Statement patterns that use one of the special geo-spatial predicates supported by OWLIM-SE are treated differently by the query engine. The following special syntax is supported when evaluating SPARQL queries (the descriptions all use the namespace `omgeo: <http://www.ontotext.com/owlim/geo#>`):

Construct	Nearby (lat long distance)
Syntax	<code>?point omgeo:nearby(?lat ?long ?distance)</code>

Description	<p>This statement pattern will evaluate to true if the following constraints hold:</p> <ul style="list-style-type: none"> • ?point geo:lat ?plat . • ?point geo:long ?plong . • Shortest great circle distance from (?plat, ?plong) to (?lat, ?long) <= ?distance <p>Such a construction will use the geo-spatial indices to find bindings for ?point that lie within the defined circle. Constants are allowed for any of ?lat ?long ?distance, where latitude and longitude are specified in decimal degrees and distance is specified in either kilometres ('km' suffix) or miles ('mi' suffix). If the units are not specified, then 'km' is assumed.</p>
Restrictions	<p>Latitude is limited to the range -90 (South) to +90 (North)</p> <p>Longitude is limited to the range -180 (West) to +180 (East)</p>
Examples	<p>Find the names of airports that are within 50 miles of Seoul:</p> <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT distinct ?airport WHERE { ?base geo-ont:name "Seoul" . ?base geo-pos:lat ?latBase . ?base geo-pos:long ?longBase . ?link omgeo:nearby(?latBase ?longBase "50mi") . ?link geo-ont:name ?airport . ?link geo-ont:featureCode geo-ont:S.AIRP . }</pre>

Construct	Within (rectangle)
Syntax	?point omgeo:within(?lat ₁ ?long ₁ ?lat ₂ ?long ₂)
Description	<p>This statement pattern is used to test/find points that lie within the rectangle specified by diagonally opposite corners ?lat1 ?long1 and ?lat2 ?long2. The corners of the rectangle must be either constants or bound values. It will evaluate to true if the following constraints hold:</p> <ul style="list-style-type: none"> • ?point geo:lat ?plat . • ?point geo:long ?plong . • ?lat₁ <= ?plat <= ?lat₂ • ?long₁ <= ?plong <= ?long₂ <p>Note that the corners must be specified most westerly and southerly (first) and most northerly and easterly (second). Proper account is taken for rectangles that cross the +/-180 degree meridian. Constants are allowed for any of ?lat₁ ?long₁ ?lat₂ ?long₂, where latitude and longitude are specified in decimal degrees. If ?point is unbound then bindings for all points within the rectangle will be produced.</p>
Restrictions	<p>Latitude is limited to the range -90 (South) to +90 (North)</p> <p>Longitude is limited to the range -180 (West) to +180 (East)</p> <p>Rectangle vertices must be specified in the order lower-left followed by upper-right</p>
Examples	<p>Find tunnels lying within a rectangle enclosing Tirol, Austria:</p> <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT ?feature ?lat ?long WHERE { ?link omgeo:within(45.85 9.15 48.61 13.18) . ?link geo-ont:featureCode geo-ont:R.TNL . ?link geo-ont:name ?feature . ?link geo-pos:lat ?lat . ?link geo-pos:long ?long . }</pre>

Construct	Within (polygon)
Syntax	<code>?point omgeo:within(?lat₁ ?long₁ ... ?lat_n ?long_n)</code>
Description	<p>This statement pattern is used to test/find points that lie within the polygon whose vertices are specified by three or more latitude/longitude pairs. The values for the vertices must be either constants or bound values. It will evaluate to true if the following constraints hold:</p> <ul style="list-style-type: none"> • <code>?point geo:lat ?plat</code> . • <code>?point geo:long ?plong</code> . • the position <code>?plat ?plong</code> is enclosed by the polygon <p>The polygon is closed automatically if the first and last vertices do not coincide. The vertices must be constants or bound values. Coordinates are specified in decimal degrees. If ?point is unbound then bindings for all points within the polygon will be produced.</p>
Restrictions	<p>Latitude is limited to the range -90 (South) to +90 (North)</p> <p>Longitude is limited to the range -180 (West) to +180 (East)</p>
Examples	<p>Find caves in the sides of cliffs lying within a polygon approximating the shape of England:</p> <pre> PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#> PREFIX geo-ont: <http://www.geonames.org/ontology#> PREFIX omgeo: <http://www.ontotext.com/owlim/geo#> SELECT ?feature ?lat ?long WHERE { ?link omgeo:within("51.45" "-2.59" "54.99" "-3.06" "55.81" "-2.03" "52.74" "1.68" "51.17" "1.41") . ?link geo-ont:featureCode geo-ont:S.CAVE . ?link geo-ont:name ?feature . ?link geo-pos:lat ?lat . ?link geo-pos:long ?long . }</pre>

Extension query functions

At present there is just one SPARQL extension function.

Function	Distance function
Syntax	<code>double omgeo:distance(?lat₁, ?long₁, ?lat₂, ?long₂)</code>
Description	This SPARQL extension function computes the distance between two points in kilometres and can be used in FILTER and ORDER BY clauses.
Restrictions	<p>Latitude is limited to the range -90 (South) to +90 (North)</p> <p>Longitude is limited to the range -180 (West) to +180 (East)</p>

Examples

Find all the airports within 80 miles of Bournemouth and filter out those that are more than 80 kilometres from Brize Norton, order the results with the closest to Brize Norton first:

```
PREFIX geo-pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX geo-ont: <http://www.geonames.org/ontology#>
PREFIX omgeo: <http://www.ontotext.com/owlim/geo#>

SELECT distinct ?airport_name
WHERE {
  ?a1 geo-ont:name "Bournemouth" .
  ?a1 geo-pos:lat ?lat1 .
  ?a1 geo-pos:long ?long1 .
  ?airport omgeo:nearby(?lat1 ?long1 "80mi" ) .
  ?airport geo-ont:name ?airport_name .
  ?airport geo-ont:featureCode geo-ont:S.AIRP .
  ?airport geo-pos:lat ?lat2 .
  ?airport geo-pos:long ?long2 .
  ?a2 geo-ont:name "Brize Norton" .
  ?a2 geo-pos:lat ?lat3 .
  ?a2 geo-pos:long ?long3 .
  FILTER( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) < 80 )
}
ORDER BY ASC( omgeo:distance(?lat2, ?long2, ?lat3, ?long3) )
```

Implementation details

Knowledge of the implementation's algorithms and assumptions will allow users to make the best use of the OWLIM-SE geo-spatial extensions. The following points are significant and can affect the expected behaviour during query answering:

- Spherical Earth – the current implementation treats the Earth as a perfect sphere with a radius of 6371.009km;
- Only 2-Dimensional points are supported, i.e. there is no special handling of geo:alt (metres above the reference surface of the Earth);
- All latitude and longitude values must be specified using decimal degrees, where East and North are positive and $-90 \leq \text{latitude} \leq +90$ and $-180 \leq \text{longitude} \leq +180$;
- Distances must be in units of kilometres (suffix 'km') or statute miles (suffix 'mi'). If the suffix is omitted, kilometres are assumed;
- `omgeo:within(rectangle)` construct uses a 'rectangle' whose edges are lines of latitude and longitude, so the north-south distance is constant and the rectangle described forms a band around the Earth that starts and stops at the given longitudes;
- `omgeo:within(polygon)` joins vertices with straight lines on a cylindrical projection of the Earth tangential to the equator. A straight line starting at the point under test and continuing East out of the polygon is examined to see how many polygon edges it intersects. If the number of intersections is even then the point is outside the polygon, if the number of intersections is odd, the point is inside the polygon. With the current algorithm, the order of vertices is not relevant (clockwise or anticlockwise);
- `omgeo:within()` may not work correctly when the region (polygon or rectangle) spans the ± 180 meridian;
- `omgeo:nearby()` uses the great circle distance between points.

OWLIM-SE RDF Rank

RDF Rank is an algorithm that identifies the more important or more popular entities in the repository by examining their interconnectedness. The popularity of entities can then be used to order query results in a similar way to internet search engines, such as how Google orders search results using PageRank <http://en.wikipedia.org/wiki/PageRank>.

The RDF Rank component computes a numerical weighting for all the nodes in the entire RDF graph stored in the repository, including URIs, blank nodes and literals. The weights are floating point numbers with values between 0 and 1 that can be interpreted as a measure of a node's relevance/popularity.

Since the values range from 0 to 1, the weights can be used for sorting a result set (the lexicographical order works fine even if the rank literals are interpreted as plain strings). Here is an example SPARQL query that uses RDF rank for sorting results by their popularity:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
PREFIX opencyc-en: <http://sw.opencyc.org/2008/06/10/concept/en/>
SELECT * WHERE {
  ?Person a opencyc-en:Entertainer .
  ?Person rank:hasRDFRank ?rank .
}
ORDER BY DESC(?rank) LIMIT 100
```

As seen in the example query, RDF Rank weights are made available via a special system predicate. Triple patterns with the predicate <http://www.ontotext.com/owlim/RDFRank#hasRDFRank> are handled specially by OWLIM, where the object of the statement pattern is bound to a literal containing the RDF Rank of the subject.

In order to use this mechanism the RDF ranks for the whole repository must be computed in advance. This is done by executing a series of SPARQL ASK queries to parameterise the weighting algorithm, followed by a query that triggers the computation itself.

Parameter	Maximum iterations
Predicate	http://www.ontotext.com/owlim/RDFRank#maxIterations
Description	Sets the maximum number of iterations of the algorithm over all entities in the repository.
Default	20
Example	PREFIX rank: < http://www.ontotext.com/owlim/RDFRank# > ASK { rank:maxIterations rank:setParam "16" . }

Parameter	Epsilon
Predicate	http://www.ontotext.com/owlim/RDFRank#epsilon
Description	Used to terminate the weighting algorithm early when the total change of all RDF Rank scores has fallen below this value.
Default	0.01
Example	PREFIX rank: < http://www.ontotext.com/owlim/RDFRank# > ASK { rank:epsilon rank:setParam "0.05" . }

To trigger the computation of the RDF Rank weights, use the following query:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:compute _:b2. }
```

The computed weights can be exported to an external file using a query of this form:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:export "/home/user1/rdf_ranks.txt" . }
```

The query will return true if the export was successful, false otherwise. If the export failed then an error message will be recorded in the log file.

Lastly, when using [RDF Priming](#), the RDF Rank weights can be used as the initial action values. To set this up, use the following query:

```
PREFIX rank: <http://www.ontotext.com/owlim/RDFRank#>
ASK { _:b1 rank:ranksAsWeights _:b2 . }
```

OWLIM-SE Notifications

- [Local Notifications](#)
- [Remote notifications](#)
 - [Using remote notifications](#)
 - [Remote Notification Configuration](#)

Local Notifications

Notifications are a publish/subscribe mechanism for registering and receiving events from an OWLIM-SE repository whenever triples matching a certain graph pattern are inserted or removed. The Sesame API provides such a mechanism, where a `RepositoryConnectionListener` can be notified of changes to a `NotifyingRepositoryConnection`. However the OWLIM-SE notifications API works at a lower level and uses the internal raw entity IDs for subject, predicate, object instead of Java objects. The benefit of this is that a much higher performance is possible. The downside is that the client must do a separate lookup up to get the actual entity values and because of this, the notification mechanism will only work when the client is running inside the same JVM as the repository instance. See the next section for the remote notification mechanism.

The user of the notifications API registers for notifications by providing a SPARQL query. The SPARQL query is interpreted as a plain graph pattern by ignoring all the more complicated SPARQL constructs like `FILTER`, `OPTIONAL`, `DISTINCT`, `LIMIT`, `ORDER BY`, etc. Therefore the SPARQL query is interpreted as a complex graph pattern involving triple patterns combined by means of joins and unions at any level. The order of the triple patterns is not significant.

Here is an example how to register for notifications based on a given SPARQL query:

```
AbstractRepository rep =
    ((OwlSchemaRepository)owlSail).getRepository();
EntityPool ent = ((OwlSchemaRepository)owlSail).getEntities();
String query = "SELECT * WHERE { ?s rdf:type ?o }";
SPARQLQueryListener listener =
    new SPARQLQueryListener(query, rep, ent) {
        public void notifyMatch(int subj, int pred, int obj, int context) {
            System.out.println("Notification on subject: " + subj);
        }
    }
rep.addListener(listener);    // start receiving notifications
...
rep.removeListener(listener); // stop receiving notifications
```

In the example code, the caller would be asynchronously notified about incoming statements matching the pattern `?s rdf:type ?o`. In general, notifications will be sent for all incoming triples that contribute to a solution of the query. The integer parameters in the `notifyMatch` method can be mapped to values using the `EntityPool` object. Furthermore, any statements inferred from newly inserted statements will also be subject to handling by the notification mechanism, i.e. new implicit statements will also be notified to clients when the requested triple pattern matches.

The subscriber should not rely on any particular order or distinctness of the statement notifications. Duplicate statements might be delivered in response to a graph pattern subscription in an order not even bound to the chronological order of the statements insertion in to the underlying triple store.

The purpose of the notification services is to enable the efficient and timely discovery of newly added RDF data. Therefore it should be treated as a mechanism for giving the client a hint that certain new data is available and not as an asynchronous SPARQL evaluation engine.

Remote notifications

OWLIM's remote notification mechanism provides filtered statement add/remove and transaction begin/end notifications for a local or a remote OWLIM-SE repository. Subscribers for this mechanism use patterns of subject, predicate and object (with wildcards) to filter the statement notifications. JMX is used internally as a transport mechanism.

Using remote notifications

Registering and deregistering for notifications is achieved through the `NotifyingOwlConnection` class, which wraps a `RepositoryConnection` object connected to an OWLIM repository and provides an API to add/remove notification listeners of type `RepositoryNotificationsListener`. Here is a simple example of the API usage:

```

RepositoryConnection conn = null;
// initialize repository connection to OWLIM ...

RepositoryNotificationsListener listener = new RepositoryNotificationsListener() {
    @Override
    public void addStatement(Resource subject, URI predicate,
        Value object, Resource context, boolean isExplicit, long tid) {
        System.out.println("Added: " + subject + " " + predicate + " " + object);
    }
    @Override
    public void removeStatement(Resource subject, URI predicate,
        Value object, Resource context, boolean isExplicit, long tid) {
        System.out.println("Removed: " + subject + " " + predicate + " " + object);
    }
    @Override
    public void transactionStarted(long tid) {
        System.out.println("Started transaction " + tid);
    }
    @Override
    public void transactionComplete(long tid) {
        System.out.println("Finished transaction " + tid);
    }
};

NotifyingOwlimConnection nConn = new NotifyingOwlimConnection(conn);
URIImpl ex = new URIImpl("http://example.com/");

// subscribe for statements with 'ex' as subject
nConn.subscribe(listener, ex, null, null);

// note that this could be any other connection to the same repository
conn.add(ex, ex, ex);
conn.commit();
// statement added should have been printed out

// stop listening for this pattern
nConn.unsubscribe(listener);

```



Note that the `transactionStarted()` and `transactionComplete()` events are not bound to any statement, they are dispatched to all subscribers, no matter what they are subscribed for. This means that pairs of start/complete events can be detected by the client without receiving any statement notifications in between.

The above example will work when the OWLIM repository is initialized in the same JVM that runs the example (local repository). If a remote repository is used (e.g. HTTPRepository) the notifying repository connection should be initialized differently:

```

NotifyingOwlimConnection nConn =
    new NotifyingOwlimConnection(conn, host, port);

```

where `host` (String) and `port` (int) are the host name of the remote machine where the repository resides and the port number of the JMX service in the repository JVM. The other part of the example remains valid for the remote case. The repository connection used to initialize a `NotifyingOwlimConnection` instance could be a `ReplicationClusterConnection` in which case notifications will work with an OWLIM-Enterprise master node (transparently to the user) - no changes on the client side are required.

Remote Notification Configuration

For remote notifications, where the subscriber and the repository are running in different JVM instances (possibly on different hosts), a JMX remote service should be configured in the repository JVM. This is done by adding the following parameters to the JVM command line:

```

-Dcom.sun.management.jmxremote.port=1717
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false

```

If the repository is running inside a servlet container, then these parameters must be passed to the JVM that runs the container and OWLIM. For Tomcat, this can be done using the `JAVA_OPTS` or `CATALINA_OPTS` environment variable.

The port number used should be exactly the port number that is passed to the `NotifyingOwlimConnection` constructor (as in the example above). One should make sure that the specified port (e.g. 1717) is accessible remotely, i.e. no firewalls or NAT redirection prevent

access to it.

In an OWLIM-Enterprise cluster setup, all the worker nodes should have their JMX configured properly in order to enable notifications for the whole cluster. The master node assumes that each worker is exposing its JMX service on port 1717 but this can be overridden when nodes are added to the cluster (the third parameter to `addClusterNode()` operation is the JMX service port of that node) or by editing the `cluster.properties` configuration file and adding the following parameter:

```
jmxport<N> = <PORTN>
```

where N is the consecutive number of the node we want to configure and PORTN is the port number of that node's JMX service. Cluster workers should also have their `com.sun.management.jmxremote.*` JVM parameters properly configured. OWLIM-Enterprise cluster master nodes will therefore be controlled and emit notifications using the same JMX port number.

OWLIM-SE Query Behaviour

- [SPARQL compliance](#)
- [Named graphs](#)
 - [The default SPARQL dataset](#)
- [Managing Explicit and Implicit Statements](#)
- [Specifying the dataset programmatically](#)
- [Accessing internal identifiers for entities](#)
 - [Examples](#)
- [Other special query behaviour](#)

This section of the user guide is intended to provide all relevant information regarding SPARQL query processing in OWLIM. The following paragraphs will detail the standards supported, implementation specific behaviour permitted by the standards and extensions that add new functionality.

SPARQL compliance

OWLIM supports the following SPARQL specifications:

- [SPARQL 1.1 Protocol for RDF](#) (working draft 26 January 2010)
- [SPARQL 1.1 Query](#) (working draft 12th May 2011)
- [SPARQL 1.1 Update](#) (working draft 12th May 2011)
- [SPARQL 1.1 Federation](#) (working draft 1st June 2010)

Note that these are working drafts and have not yet reached recommendation status with the [W3C](#).

SPARQL 1.1 Protocol for RDF defines the means for transmitting SPARQL queries to a SPARQL query processing service and returning the query results to the entity that requested them.

SPARQL 1.1 Query provides more powerful query constructions compared to SPARQL 1.0. It adds:

- Aggregates
- Subqueries
- Negation
- Expressions in the SELECT clause
- Property Paths
- Assignment
- An expanded set of functions and operators

SPARQL 1.1 Update provides a means to change the state of the database using a query-like syntax. SPARQL Update has similarities to SQL INSERT INTO, UPDATE WHERE and DELETE FROM behaviour.

SPARQL 1.1 Federation provides extensions to the query syntax for executing distributed queries over any number of SPARQL endpoints. This new feature from Sesame 2.6 is very powerful, but must be used with caution. The following example finds resources in the second SPARQL endpoint that have a similar `rdfs:label` to the `rdfs:label` of `<http://dbpedia.org/resource/Vaccination>` in the first SPARQL endpoint:

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

SELECT ?endpoint2_id {
  SERVICE <http://faraway_endpoint.org/sparql>
  {
    ?endpoint1_id rdfs:label ?l1 .
    FILTER( lang(?l1) = "en" )
  }
  SERVICE <http://remote_endpoint.com/sparql>
  {
    ?endpoint2_id rdfs:label ?l2 .
    FILTER( str(?l2) = str(?l1) )
  }
}
BINDINGS ?endpoint1_id
{ ( <http://dbpedia.org/resource/Vaccination> ) }
```

However, such a query is very inefficient, because no intermediate bindings are passed between endpoints. Instead, both sub-queries execute independently, requiring the second sub-query to return all `X rdfs:label Y` statements that it stores. These are then joined locally to the (likely much smaller) results of the first sub-query.

Named graphs

An RDF database can store collections of RDF statements (triples) in separate graphs identified (named) by a URI. A group of statements with a unique name is called a named graph. An RDF database has one more graph that does not have a name and this is called the 'default graph'.

The SPARQL query syntax provides a means to execute queries across default and named graphs using FROM and FROM NAMED clauses. These clauses are used to build an 'RDF Dataset' that identifies what statements the SPARQL query processor will use to answer a query. The dataset contains a default graph and named graphs and is constructed as follows:

- **FROM uri** - brings statements from the database's graph identified by 'uri' in to the dataset's default graph, i.e. the statements 'lose' their graph name
- **FROM NAMED uri** - brings the statements from database's graph identified by 'uri' in to the dataset, i.e. the statements keep their graph name

If either FROM or FROM NAMED are used then the database's default graph is no longer used as input for processing this query. In effect, the combination of FROM and FROM NAMED clauses exactly defines the dataset. This is somewhat bothersome, as it precludes the possibility, for instance, of executing a query over just one named graph and the default graph. However, there is a programmatic way to get around this limitation that is described below.

The default SPARQL dataset

The SPARQL specification does not define what happens when no FROM or FROM NAMED clauses are present in a query, i.e. it does not define how a SPARQL processor should behave when no dataset is defined. In this situation, implementations are free to construct the default dataset in any way they please.

OWLIM constructs the default dataset as follows:

- The dataset's default graph contains the merge of the database's default graph AND all the database's named graphs
- The dataset contains all named graphs from the database

This means that if a statement `ex:x ex:y ex:z` exists in the database in graph `ex:g` then the following query patterns will behave as follows:

Query	Bindings
<code>SELECT * { ?s ?p ?o }</code>	<code>?s=ex:x ?p=ex:y ?o=ex:z</code>
<code>SELECT * { GRAPH ?g { ?s ?p ?o } }</code>	<code>?s=ex:x ?p=ex:y ?o=ex:z ?g=ex:g</code>

in other words, the triple `ex:x ex:y ex:z` will appear to be in both the default graph and the named graph `ex:g` at the same time.

There are several reasons for this behaviour:

1. It provides an easy way to execute a triple pattern query over all stored RDF statements
2. It allows all named graph names to be discovered, i.e. with this query: `SELECT ?g { GRAPH ?g { ?s ?p ?o } }`

Managing Explicit and Implicit Statements

Explicit statements: Statements that have been inserted by the user in to a database, using SPARQL Update, the Sesame API or the 'imports' configuration parameter are flagged as being 'explicit'. Explicit statements can exist in the database's default graph and named graphs.

Implicit statements: Statements that have been created as a result of inference are flagged as being implicit and are stored ONLY in the database's default graph.

Therefore, the database's default graph can contain a mixture of explicit and implicit statements. The Sesame API provides a flag called 'includeInferred' that can help in some situations. This flag is passed to several API methods and when set to false will cause only explicit statements to be iterated or returned. When this flag is set to true, then both explicit and implicit statements are iterated or returned.

OWLIM provides extensions for providing more control over the processing of explicit and implicit statements. These extensions allow the selection of explicit, implicit, or both for query answering and also provides a mechanism to identify which statements are explicit and which are implicit. This is achieved by using some 'pseudo-graph' names in FROM and FROM NAMED clauses that cause certain flags to be set. The details are as follows:

Clause	Behaviour
<code>FROM < http://www.ontotext.com/explicit ></code>	The dataset's default graph will include only explicit statements from the database's default graph
<code>FROM < http://www.ontotext.com/implicit ></code>	The dataset's default graph will include only inferred statements from the database's default graph

<pre>FROM NAMED < http://www.ontotext.com/explicit ></pre>	<p>The dataset will contain a named graph called http://www.ontotext.com/explicit that contains only explicit statements from the database's default graph, i.e. quad patterns such as {GRAPH ?g {?s ?p ?o}} will bind to explicit statements from the database's default graph with a graph name of <</p> <p>http://www.ontotext.com/explicit</p> <p>></p>
<pre>FROM NAMED < http://www.ontotext.com/implicit ></pre>	<p>The dataset will contain a named graph called http://www.ontotext.com/implicit that contains only implicit statements from the database's default graph</p>

Note that these are only flags and do not affect the construction of the default dataset in the sense that using any combination of the above will still result in the dataset containing all the named graphs from the database. All that is changed is which statements appear in the dataset's default graph and whether any extra named graphs (explicit or implicit) appear.

Specifying the dataset programmatically

The Sesame API provides an interface `Dataset` and an implementation class `DatasetImpl` for defining the dataset for a query by providing the URIs of named graphs and adding them to the 'default graphs' and 'named graphs' members. This permits the use of 'null' to be used to identify the default database graph (or 'null context' to use Sesame terminology).

```
DatasetImpl dataset = new DatasetImpl();
dataset.addDefaultGraph(null);
dataset.addNamedGraph(valueFactory.createURI("http://example.com/g1"));
```

This dataset can then be passed to queries or updates, e.g.

```
TupleQuery query = connection.prepareTupleQuery(QueryLanguage.SPARQL, queryString);
query.setDataset(dataset);
```

Accessing internal identifiers for entities

Internally, OWLIM uses integer identifiers (IDs) to index all entities (URIs, blank nodes and literals). Statement indices are made up of these IDs and a large data structure is used to map from ID to entity value and back. There are occasions, e.g. when interfacing to application infrastructure, when having access to these internal IDs can improve the efficiency of data structures external to OWLIM by allowing them to be indexed by an integer value rather than a full URI.

This section introduces a special OWLIM predicate and function that provide access to these internal IDs. The datatype of internal IDs is [<http://www.w3.org/2001/XMLSchema#long>](http://www.w3.org/2001/XMLSchema#long).

Predicate	<http://www.ontotext.com/owlim/entity#id>
Description	Map between entity and internal ID
Example	<p>Select all entities and their IDs:</p> <p>PREFIX ent: <http://www.ontotext.com/owlim/entity#></p> <pre>SELECT * WHERE { ?s ent:id ?id } ORDER BY ?id</pre>

Function	<http://www.ontotext.com/owlim/entity#id>
Description	Return an entity's internal ID
Example	<p>Select all statements and order them by the internal ID of the object values:</p> <p>PREFIX ent: <http://www.ontotext.com/owlim/entity#></p> <pre>SELECT * WHERE { ?s ?p ?o . } order by ent:id(?o)</pre>

Examples

- Enumerate all the entities and bind the nodes to ?s and their IDs to ?id, order by ?id:

```
select * where {
  ?s <http://www.ontotext.com/owlim/entity#id> ?id
} order by ?id
```

- Enumerate all non-literals and bind the nodes to ?s and their IDs to ?id, order by ?id:

```
SELECT * WHERE {
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .
  FILTER (!isLiteral(?s)) .
} ORDER BY ?id
```

- Find the internal IDs of subjects of statements with specific predicate and object values:

```
SELECT * WHERE {
  ?s <http://test.org#Pred1> "A literal".
  ?s <http://www.ontotext.com/owlim/entity#id> ?id .
} ORDER BY ?id
```

- Find all statements where the object has the given internal ID by using an explicit, un-typed value as the ID (the "115" used as object in the second statement pattern):

```
SELECT * WHERE {
  ?s ?p ?o.
  ?o <http://www.ontotext.com/owlim/entity#id> "115" .
}
```

- As above, but using an xsd:long datatype for the constant within a FILTER condition:

```
SELECT * WHERE {
  ?s ?p ?o.
  ?o <http://www.ontotext.com/owlim/entity#id> ?id .
  FILTER (?id="115"^^<http://www.w3.org/2001/XMLSchema#long>) .
} ORDER BY ?o
```

- Find the internal IDs of subject and object entities for all statements:

```
SELECT * WHERE {
  ?s ?p ?o.
  ?s <http://www.ontotext.com/owlim/entity#id> ?ids.
  ?o <http://www.ontotext.com/owlim/entity#id> ?ido.
}
```

- Retrieve all statements where the ID of the subject is equal to "115"^^xsd:long, by providing an internal ID value within a filter expression:

```
SELECT * WHERE {
  ?s ?p ?o.
  FILTER ((<http://www.ontotext.com/owlim/entity#id>(?s)) =
"115"^^<http://www.w3.org/2001/XMLSchema#long>).
}
```

- Retrieve all statements where the string-ised ID of the subject is equal to "115", by providing an internal ID value within a filter expression:

```
SELECT * WHERE {
  ?s ?p ?o.
  FILTER (str( <http://www.ontotext.com/owlim/entity#id>(?s) ) = "115").
}
```

Other special query behaviour

There are several more special graph URIs used in OWLIM-SE that can be used to control query evaluation.

Clause	Behaviour
FROM/FROM NAMED < http://www.ontotext.com/disable-sameAs >	Used to switch off the enumeration of the equivalence classes produced by owl:sameAs during triple pattern matching, which is the default behaviour, so that solutions followed by these are excluded. Its purpose is to reduce the number of results to only those that are valid for a single representative of the class (this is a rough description and not fully explanatory). For example, given a triple that matches a pattern: test:Inst rdf:type, test:SomeClass and test:Inst is owl:sameAs to test:Inst2 then, by default there would be 2 triples matching the pattern, one for test:Inst and another for test:Inst2. Using the above system graph in FROM/FROM NAMED clauses excludes such redundancies. BE AWARE that if the query uses filters over the textual representation of a node that modifier may skip some valid solutions since not all the nodes within an equivalence class will be matched against such a FILTER.
FROM/FROM NAMED < http://www.ontotext.com/count >	Will trigger the evaluation of the query so that it will give a single result in which all the variable bindings in the projection will be replaced with a plain literal holding the value of the total number of solutions of the query, i.e. the equivalent of COUNT(*) from SQL. In the case of a CONSTRUCT query in which the projection contains three variables (?subject, ?predicate, ?object), the subject and the predicate will be bound to < http://www.ontotext.com/ > and the object will hold the literal value. This is because there cannot exist a statement with literal in the place of the subject or predicate.
FROM/FROM NAMED < http://www.ontotext.com/skip-redundant-implicit >	Will trigger the exclusion of implicit statements when there exists an explicit one within a specific context(even default). Initially implemented to allow for filtering of redundant rows where the context part is not taken into account and which leads to 'duplicate' results.
FROM < http://www.ontotext.com/distinct >	Using this special graph name in DESCRIBE and CONSTRUCT queries will cause only distinct triples to be returned. This is useful when several resources are being described, where the same triple can be returned more than once, i.e. when describing its subject and its object.
FROM < http://www.ontotext.com/owlim/cluster/control-query >	Identifies the query to the OWLIM-Enterprise cluster master node as needing to be routed to all worker nodes.

OWLIM-SE Plug-in API

- [Overview](#)
- [Description of an OWLIM plug-in](#)
- [The life-cycle of a plug-in](#)
- [Repository Internals \(Statements and Entities\)](#)
- [Request-Processing Phases](#)
 - [Pre-processing](#)
 - [Pattern Interpretation](#)
 - [Post-processing](#)
- [Putting It All Together: An example Plug-in](#)
- [Making a Plug-in Configurable](#)
- [Accessing other plug-ins](#)

Overview

The OWLIM Plug-in API is a framework and a set of public classes and interfaces that allow developers to extend OWLIM in many useful ways. These extensions are bundled into plug-ins that OWLIM discovers during its initialisation phase and then uses to delegate parts of its query processing tasks. The plug-ins are given low-level access to OWLIM repository data that enables them to do their job efficiently. The plug-ins are discovered via the Java service discovery mechanism which enables dynamic addition/removal of plug-ins from the system without having to recompile OWLIM or change any configuration files.

This section will cover the plug-in capabilities that the framework provides and introduced mostly by example.

Description of an OWLIM plug-in

An OWLIM plug-in is a java class that implements the `com.ontotext.trree.sdk.Plugin` interface. All the public classes and interfaces of the plug-in API are located in this java package, i.e. `com.ontotext.trree.sdk`, so this package name will be omitted for the rest of this section. Here is what the Plugin interface looks like in abbreviated form:

```
public interface Plugin extends Service {
    void setStatements(Statements statements);

    void setEntities(Entities entities);

    void setOptions(SystemOptions options);

    void setDataDir(File dataDir);

    void setLogger(Logger logger);

    void initialize();

    void setFingerprint(long fingerprint);

    long getFingerprint();

    void shutdown();
}
```

Being derived from the `Service` interface means that plug-ins will be automatically discovered at run-time provided that the following conditions also hold:

- The plug-in class is located somewhere in the classpath
- It is mentioned in a `META-INF/services/com.ontotext.trree.sdk.Plugin` file in the classpath or in a jar which is in the classpath. The full class signature should be written in such a file alone on a separate line

The only method introduced by the `Service` interface is `getName()` which provides the plug-in's (service's) name. This name should be unique within a particular OWLIM repository and serves as a plug-in identifier that can be used at any time to retrieve a reference to the plug-in instance. The rest of the base `Plugin` methods will be described further in the following sections.

There are a lot more functions (interfaces) that a plug-in could implement, but these are all optional and are declared in separate interfaces. Implementing any such complementary interface is the means to announce to the system what this particular plug-in can do in addition to its mandatory plug-in responsibilities. It is then automatically used as appropriate.

The life-cycle of a plug-in

A plug-in's life-cycle is separated into several phases:

- **Discovery** - this phase is executed at repository initialisation. OWLIM searches for all plug-in services registered in `META-INF/services/com.ontotext.trree.sdk.Plugins` service registry files and constructs a single instance of each plug-in found
- **Configuration** - every plug-in instance discovered and constructed during the previous phase is then configured. During this phase plug-ins are injected with a `Logger` object that they should use for logging (`setLogger(Logger logger)`) and the path to their own data directory (`setDataDir(File dataDir)`) that they should create if needed and then use to store their data. If a plug-in doesn't need to store anything to disk then it can skip the creation of its data directory. However, if it needs to use it, it is guaranteed that this directory will be unique and available only to the particular plug-in that it was assigned to. The plug-ins are also injected `Statements` and `Entities` instances (see below) and a `SystemOptions` instance that gives plug-ins access to the system-wide configuration options and settings.
- **Initialisation** - after a plug-in has been configured the framework calls its `initialize()` method so it gets the chance to do whatever initialisation work it needs to do. It is important at this point that the plug-in has received all its configuration and low-level access to the repository data (see [Statements and Entities below](#)).
- **Request** - the plug-in participates in request processing. This phase is optional for plug-ins. It is divided into several sub-phases and each plug-in can choose to participate in any or none of these. The *request* phase not only includes the evaluation of, for instance SPARQL queries, but also SPARQL/Update requests and `getStatements` calls. Here are the sub-phases of the *request* phase:
 - **Pre-processing** - plug-ins are given the chance to modify the request before it is processed. In this phase they could also initialise a context object which will be visible till the end of the request processing (see below)
 - **Pattern interpretation** - plug-ins can choose to provide results for requested statement patterns (see below)
 - **Post-processing** - before the request results are returned to the client, plug-ins are given a chance to modify them, filter them out or even insert new results (see below).
- **Shutdown** - during repository shutdown, each plug-in is prompted to execute its own shutdown routines, free resources, flush data to disk, etc. This should be done in the `shutdown()` method.

Repository Internals (Statements and Entities)

In order to enable efficient request processing plug-ins are given low-level access to the repository data and internals. This is done through the `Statements` and `Entities` interfaces.

The `Entities` interface represents a set of RDF objects (URIs, blank nodes and literals). All such objects are termed *entities* and are given unique long identifiers. The `Entities` instance is responsible for resolving those objects from their identifiers and inversely for looking up the identifier of a given entity. Most plug-ins will process entities using their identifiers, because dealing with integer identifiers is a lot more efficient than working with the actual RDF entities they represent. The `Entities` interface is the single entry point available to plug-ins for entity management. It supports the addition of new entities, entity replacement, look-up of entity type and properties, resolving entities, listening for entity change events, etc. It is possible in an OWLIM repository to declare two RDF objects to be equivalent (e.g. by using `owl:sameAs`). In order to provide a way to use such declarations, the `Entities` interface assigns a class identifier to each entity. For newly created entities this class identifier is the same as the entity identifier. When two entities are declared equivalent one of them adopts the class identifier of the other and thus they become members of the same equivalence class. The `Entities` interface exposes the entity class identifier for plug-ins to determine which entities are equivalent. Entities within an `Entities` instance have a certain scope. There are three entity scopes:

- **Default** - entities stored in this scope are persisted to disk and can be used in statements that are also physically stored on disk. These entities have non-zero, positive identifiers and are often referred to physical entities.
- **System** - system entities have negative identifiers and are not persisted to disk. They can be used e.g. for system (or *magic*) predicates. They are available throughout the whole repository lifetime but after it is restarted they disappear and need to be re-created again should one need them.
- **Request** - entities stored in request scope, like system entities, are not persisted on disk and have negative identifiers. However, they only live in the scope of a particular request. They are not visible to other concurrent requests and disappear immediately after the request processing has finished. This scope is useful for temporary entities like literal values that are not expected to occur often (e.g. numerical values) and don't appear inside a physical statement.

The `Statements` interface represents a set of RDF statements where statement means a quadruple of *subject*, *predicate*, *object* and *context* RDF entity identifiers. Statements can be added, removed and searched for. Additionally, a plug-in can subscribe to receive statement event notifications (e.g. "statement was added").

An important abstract class which is related to OWLIM internals is `StatementIterator`. It has a single abstract method - `boolean next()` - which attempts to scroll the iterator onto the next available statement and returns true only if it succeeded. In the case of success its `subject`, `predicate`, `object` and `context` fields will be initialised with the respective components of the next statement.

Here is a brief example that puts `Statements`, `Entities` and `StatementIterator` together in order to output all literals that are related to a given URI:

Putting Statements, Entities and StatementIterator to work

```
// resolve the URI identifier
long id = entities.resolve(new URIImpl("http://example/uri"));

// retrieve all statements with this identifier in subject position
StatementIterator iter = statements.get(id, 0, 0, 0);
while (iter.next()) {
    // only process literal objects
    if (entities.getType(iter.object) == Entities.Type.LITERAL) {
        // resolve the literal and print out its value
        Value literal = entities.get(iter.object);
        System.out.println(literal.stringValue());
    }
}
```

Getting to know these interfaces should be sufficient for a plug-in developer to make full use of OWLIM repository data.

Request-Processing Phases

As already mentioned, a plug-in's interaction with each of the request-processing phases is optional. The plug-in declares if it plans to participate in any phase by implementing the appropriate interface.

Pre-processing

A plug-in willing to participate in request pre-processing should implement the `Preprocessor` interface. It looks like this:

Preprocessor.java

```
public interface Preprocessor {
    RequestContext preprocess(Request request);
}
```

The `preprocess()` method receives the request object and returns `RequestContext` instance. The `Request` instance passed as the parameter will be a different class instance depending on the type of the request (e.g. SPARQL/Update or "get statements"). The plug-in can change the request object in the necessary way and should initialise and return its context object that will be passed back to it in every other method during the request processing phase. The returned request context may be `null` and whatever it is it will only be visible to the plug-in that initialised it. It can be used to store data, visible for (and only for) this whole request, e.g. to pass data relating to two different statement patterns recognised by the plug-in. The request context will give further request processing phases access to the `Request` object reference. Plug-ins that opt to skip this phase will not have a request context and will not be able to get access to the original `Request` object.

Pattern Interpretation

This is one of the most important phases in the lifetime of a plug-in. In fact most plug-ins need to participate in exactly this phase. This is the point where request statement patterns need to get evaluated and statement results are returned. For example, consider the following SPARQL query:

Simple SPARQL query

```
SELECT * WHERE {
    ?s <http://example/predicate> ?o
}
```

There just one statement pattern inside this query: `?s <http://example/predicate> ?o`. All plug-ins that have implemented the `PatternInterpreter` interface (thus declaring that they intend to participate in the pattern interpretation phase) will be asked if they can interpret this pattern. The first one to accept it and return results for it will be used. If no plug-in interprets the pattern it will be looked up using the repository's *physical* statements, i.e. the ones persisted on disk.

Here is the `PatternInterpreter` interface:

PatternInterpreter.java

```
public interface PatternInterpreter {
    double estimate(long subject, long predicate, long object, long context, Statements statements,
        Entities entities, RequestContext requestContext);

    StatementIterator interpret(long subject, long predicate, long object, long context,
        Statements statements, Entities entities, RequestContext requestContext);
}
```

The `estimate()` and `interpret()` methods take the same arguments and are used in the following way:

- given a statement pattern (e.g. the in the SPARQL query above) all plug-ins that implement `PatternInterpreter` are asked to `interpret()` the pattern. The `subject`, `predicate`, `object` and `context` values are either the identifiers of the values in the pattern or 0 if any of them is an unbound variable. The `statements` and `entities` objects represent respectively the statements and entities that are available for this particular request. For instance, if the query contained any `FROM <http://some/graph>` clauses, the `statements` object would only provide access to statements in the defined named graphs. Similarly, the `entities` object contains entities that might be valid only for this particular request. The plug-in's `interpret()` method must return a `StatementIterator` if it intends to interpret this pattern or `null` if it refuses.
- in case the plug-in signals that it will interpret the given pattern (returns non-null value), OWLIM's query optimiser will call the plug-in's `estimate()` method in order to get an estimate on how many results will be returned by the `StatementIterator` returned by `interpret()`. This estimate need not be precise, but the better it is, the more likely the optimiser will make an efficient optimisation. There is a slight difference in the values that will be passed to `estimate()`. The statement components (e.g. `subject`) might not only be entity identifiers, but also they can be set to 2 special values:
 - `Entities.BOUND` - the pattern component is said to be bound, but its particular binding is not yet known
 - `Entities.UNBOUND` - the pattern component will not be bound
 These values should be treated as hints to the `estimate()` method to provide a better approximation of the result set size although its precise value cannot be determined before the query is actually run
- after the query has been optimised the `interpret()` method of the plug-in might be called again should any variable become bound due to the pattern reordering applied by the optimiser. Plug-ins should be prepared to expect different combinations of bound and unbound statement pattern components and return appropriate iterators.

The `requestContext` parameter is the value returned by the `preprocess()` method if one exists or `null` otherwise.

The plug-in framework also supports the interpretation of an extended type of *list* pattern. Consider the following SPARQL query:

Simple SPARQL query

```
SELECT * WHERE {
    ?s <http://example/predicate> (?o1 ?o2)
}
```

If a plug-in wants to handle such list patterns it has to implement an interface very similar to the `PatternInterpreter` interface - `ListPatternInterpreter`:

ListPatternInterpreter.java

```
public interface ListPatternInterpreter {
    double estimate(long subject, long predicate, long[] objects, long context, Statements
        statements,
        Entities entities, RequestContext requestContext);

    StatementIterator interpret(long subject, long predicate, long[] objects, long context,
        Statements statements, Entities entities, RequestContext requestContext);
}
```

It only differs by having multiple objects passed as an array of `long`'s instead of a single `long` object. The semantics of both methods is equivalent to the one in the basic pattern interpretation case.

Post-processing

There are cases when a plug-in would like to modify or otherwise filter the final results of a request. This is where the `Postprocessor` interface comes into play:

Postprocessor.java

```
public interface Postprocessor {
    BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext);

    Iterator<BindingSet> flush(RequestContext requestContext);
}
```

The `postprocess()` method is called for each binding set that is to be returned to the repository client. This method may modify the binding set and return it or alternatively return `null` in which case the binding set is removed from the result set. After a binding set is processed by a plug-in, the possibly modified binding set is passed to the next plug-in having post-processing functionality enabled. After the binding set is processed by all plug-ins (in the case where no plug-in deletes it) it is returned to the client. Finally, after all results are processed and returned, each plug-in's `flush()` method is called to introduce new binding set results in the result set. These in turn are finally returned to the client.

Putting It All Together: An example Plug-in

The example plug-in will have two responsibilities:

- it should interpret patterns like `?s <http://example.com/time> ?o` and should bind their object component to a literal containing the repository local date and time
- if a `FROM <http://example.com/time>` clause is detected in the query the result should be a single binding set in which all projected variables should be bound to a literal containing the repository local date and time

For the first part, it is clear that the plug-in should implement the `PatternInterpreter` interface. A date/time literal should be stored as a request-scope entity to avoid cluttering the repository with extra literals.

For the second requirement the plug-in must first take part in the pre-processing phase in order to inspect the query and detect the `FROM` clause. Then the plug-in must hook into the post-processing phase, where if the pre-processing phase has detected the desired `FROM` clause it should delete all query results (in `postprocess()` and return (in `flush()`) a single result containing the binding set specified by the requirements. Again, request-scoped literals should be created.

The plug-in implementation extends the `PluginBase` class that provides a default implementation of the `Plugin` methods:

Example plugin

```
public class ExamplePlugin extends PluginBase {
    private static final URI PREDICATE = new URIImpl("http://example.com/time");
    private long predicateId;

    @Override
    public String getName() {
        return "example";
    }

    @Override
    public void initialize() {
        predicateId = entities.put(PREDICATE, Entities.Scope.SYSTEM);
    }
}
```

In this basic implementation the plug-in name is defined and during initialization a single system-scope predicate is registered. It is important not to forget to register the plug-in in the `META-INF/services/com.ontotext.tree.sdk.Plugin` file in the classpath.

The next step is to implement the first of the plug-in's requirements - the pattern interpretation part:

Example plugin

```

public class ExamplePlugin extends PluginBase implements PatternInterpreter {

    // ...

    @Override
    public StatementIterator interpret(long subject, long predicate, long object, long context,
        Statements statements, Entities entities, RequestContext requestContext) {
        // ignore patterns with predicate different than the one we recognize
        if (predicate != predicateId)
            return null;

        // create the date/time literal
        long literalId = createDateTimeLiteral();

        // return a StatementIterator with a single statement to be iterated
        return StatementIterator.create(subject, predicate, literalId, 0);
    }

    private long createDateTimeLiteral() {
        Value literal = new LiteralImpl(new Date().toString());
        return entities.put(literal, Scope.REQUEST);
    }

    @Override
    public double estimate(long subject, long predicate, long object, long context, Statements
        statements,
        Entities entities, RequestContext requestContext) {
        return 1;
    }
}

```

The `interpret()` method only processes patterns with a predicate matching the desired predicate identifier. Further on it simply creates a new date/time literal (in the request scope) and places its identifier in the object position of the returned single result. The `estimate()` method always returns 1, because this is the exact size of the result set.

Finally to implement the second requirement concerning the interpretation of the `FROM` clause:

Example plugin, pre- and post-processing

```
public class ExamplePlugin extends PluginBase implements PatternInterpreter, Preprocessor,
Postprocessor {
    private static class Context implements RequestContext {
        private Request theRequest;
        private BindingSet theResult;

        public Context(BindingSet result) {
            theResult = result;
        }
        @Override
        public Request getRequest() {
            return theRequest;
        }
        @Override
        public void setRequest(Request request) {
            theRequest = request;
        }
        public BindingSet getResult() {
            return theResult;
        }
    }

    // ...

    @Override
    public RequestContext preprocess(Request request) {
        if (request instanceof QueryRequest) {
            QueryRequest queryRequest = (QueryRequest) request;
            Dataset dataset = queryRequest.getDataset();
            if ((dataset != null && dataset.getDefaultGraphs().contains(PREDICATE))) {
                // create a date/time literal
                long literalId = createDateTimeLiteral();
                Value literal = entities.get(literalId);
                // prepare a binding set with all projected variables set to the date/time literal value
                MapBindingSet result = new MapBindingSet();
                if (queryRequest.getTupleExpr() instanceof Projection) {
                    Projection projection = (Projection) queryRequest.getTupleExpr();
                    for (String bindingName : projection.getBindingNames()) {
                        result.addBinding(bindingName, literal);
                    }
                }
                return new Context(result);
            }
        }
        return null;
    }

    @Override
    public BindingSet postprocess(BindingSet bindingSet, RequestContext requestContext) {
        // if we have found the special FROM clause we filter out all results
        return requestContext != null ? null : bindingSet;
    }

    @Override
    public Iterator<BindingSet> flush(RequestContext requestContext) {
        // if we have found the special FROM clause we return the special binding set
        BindingSet result = ((Context) requestContext).getResult();
        return requestContext != null ? new SingletonIterator<BindingSet>(result) : null;
    }
}
```

The plug-in provides the custom implementation of the `RequestContext` interface, which can hold a reference to the desired single `BindingSet` with the date/time literal bound to every variable name in the query projection. The `postprocess()` method filters out all results if the `requestContext` is non-null (i.e. if the `FROM` clause was detected by `preprocess()`). Finally `flush()` returns a singleton iterator containing the desired binding set in the required case and returns nothing otherwise.

Making a Plug-in Configurable

Plug-ins are expected to require configuring. There are two ways for OWLIM plug-ins to receive their configuration. The first practice is to

define magic system predicates that can be used to pass some configuration values to the plug-in through a query at run-time. This approach is appropriate whenever the configuration should change from one plug-in usage scenario to another, i.e. when there are no globally valid parameters for the plug-in. However, in many cases the plug-in behaviour has to be configured "globally" and then the plug-in framework provides a suitable mechanism through the `Configurable` interface.

A plug-in implements the `Configurable` interface to announce its configuration parameters to the system. This allows it to read parameter values during initialization from the repository configuration and have them merged with all other repository parameters (accessible through the `SystemOptions` instance passed during the *configuration* phase).

This is the `Configurable` interface:

Configurable.java
<pre>public interface Configurable { public String[] getParameters(); }</pre>

The plug-in needs to enumerate its configuration parameter names. The example plug-in will be extended with the ability to define the name of special predicate it uses. The parameter is called `predicate-uri` and it should accept a URI value.

Example plugin, configuration
<pre>public class ExamplePlugin extends PluginBase implements PatternInterpreter, Preprocessor, Postprocessor, Configurable { private static final String DEFAULT_PREDICATE = "http://example.com/time"; private static final String PREDICATE_PARAM = "predicate-uri"; // ... @Override public String[] getParameters() { return new String[] { PREDICATE_PARAM }; } // ... @Override public void initialize() { // get the configured predicate URI, falling back to our default if none was found String predicate = options.getParameter(PREDICATE_PARAM, DEFAULT_PREDICATE); predicateId = entities.put(new URIImpl(predicate), Entities.Scope.SYSTEM); } // ... }</pre>

Now that the plug-in parameter has been declared, it can be configured either by adding the <http://www.ontotext.com/tree/owlim#predicate-uri> parameter to the OWLIM configuration or by setting a Java system property using `-Dpredicate-uri` parameter for the JVM running OWLIM.

There are also a special kind of configuration parameters called "memory" parameters. These are parameters that are used to configure the amount of memory available for the plug-in to use. If a plug-in has such parameters it should use the `MemoryConfigurable` interface:

MemoryConfigurable
<pre>public interface MemoryConfigurable { public String[] getMemoryParameters(); public void setMemoryParameter(String name, long bytes); }</pre>

The `getMemoryParameters()` method enumerates the names of the plug-in's memory parameters in a similar way to `Configurable.getParameters()`. During the configuration phase, the plug-in's `setMemoryParameter()` method will be called once for each such parameter with its respective configured value in bytes. The parameters defined as memory parameters can be given values like "1g" or "300M", but such values will be interpreted and converted to bytes.

A special property of the memory parameters is that they can be configured in a group. OWLIM accepts a parameter called `cache-memory`. This parameter accumulates the values of a group of other parameters: `tuple-index-memory`, `fts-memory` and `predicate-memory`. Declaring a memory parameter automatically adds it in the group of parameters accumulated by `cache-memory`. What is good about this approach is that if `cache-memory` is configured to some amount and any of the grouped memory parameters is

not configured (unknown), the amount configured for `cache-memory` is divided among all unknown memory parameters thus providing the user with a simple way to control the memory requirements of many plug-ins using a single parameter. For instance, if `cache-memory` is configured to "100m", `tuple-index-memory` to "20m", there are no predicate lists configured (which automatically disables the `predicate-memory` parameter) and there are 4 memory parameters declared by several plug-ins which weren't explicitly configured. The effect of such a setup would be that 80M (100M - 20M) will be divided among the 4 memory parameters and each of them will be set to 20M. This value is then reported to the plug-ins in bytes using their `setMemoryParameter()` method.

Accessing other plug-ins

Plug-ins are able to make use of the functionality of other plug-ins. For example, the Lucene-based full-text search plug-in can make use of the rank values provided by the RDFRank plug-in to facilitate query result scoring and ordering. This is not a matter of re-using program code (e.g. in a jar with common classes), rather it is about re-using data. The mechanism to do this allows plug-ins to obtain references to other plug-in objects by knowing their names. To achieve this they only need to implement the `PluginDependency` interface:

```

PluginDependency.java

public interface PluginDependency {
    public void setLocator(PluginLocator locator);
}

```

They are then injected an instance of the `PluginLocator` interface (during configuration phase) that does the actual plug-in discovery for them:

```

PluginLocator.java

public interface PluginLocator {
    public Plugin locate(String name);
}

```

Having a reference to another plug-in is all that is needed to call its methods directly and make use of its services.

OWLIM-SE Experimental Features

- [RDF Priming](#)
 - [RDF Priming Configuration](#)
 - [Controlling RDF Priming](#)
 - [RDF Priming Example](#)

RDF Priming

RDF Priming is a technique that selects a subset of available statements for use as the input to query answering. It is based upon the concept of 'spreading activation' as developed in cognitive science. RDF Priming is a scalable and customisable implementation of the popular connectionist method on top of RDF graphs that allows for the "priming" of large datasets with respect to concepts relevant to the context and to the query. It is controlled using SPARQL ASK queries. This section provides an overview of the mechanism and explains the necessary SPARQL queries used to manage and set up RDF Priming.

RDF Priming Configuration

To enable RDF Priming over the repository, the `repository-type` configuration parameter should be set to `weighted-file-repository`.

The current implementation of RDF Priming does not store activation values, which means that they are only available at runtime and are lost when the repository is shutdown. However, they can be exported and imported using the special query directives shown below.

Another side effect is that the activation values are global, because they stored within the shared Entity pool.

The initialization and management of the RDF Priming module is achieved by performing SPARQL ASK queries.

Controlling RDF Priming

RDF Priming is controlled using SPARQL ASK queries, which allows all the parameters and default values to be set. These queries use special system predicates, which are described below:

Function	Enable Activation Spreading
Predicate	<code>http://www.ontotext.com/owlim/RDFPriming#enableSpreading</code>
Description	Used to enable or disable the RDF Priming module. The Object value of the statement pattern should be a Literal whose value is either "true" or "false"
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK {_:b1 prim:enableSpreading "true".}

Function	Set Activation Decay
Predicate	<code>http://www.ontotext.com/owlim/RDFPriming#decayActivations</code>
Description	Used to alter all the activation values for the nodes in the RDF graph by multiplying them by a factor specified as a Literal in the Object position of the Statement pattern of the query. The following example will reset all the activation values to zero by multiplying them by "0.0"
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK {_:b1 prim:decayActivations "0.0".}

Function	Trigger Activation Spreading Cycle
Predicate	<code>http://www.ontotext.com/owlim/RDFPriming#spreadActivation</code>
Description	Used to trigger an Activation spreading cycle that starts from the nodes that were scheduled for activation for this round. No special values are required for the Subject or Object part of the statement pattern – blank nodes suffice
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK {_:b1 prim:spreadActivation _:b2.}

Function	Set Statement Weight
-----------------	-----------------------------

Predicate	http://www.ontotext.com/owlim/RDFPriming#assignWeight
Description	Used to set a non-default weight factor for statements with a specific predicate. The Subject of the Statement pattern is the predicate to which the new value should be set. The Object of the pattern is the new weight value as a Literal. The example query sets 0.5 as a weight factor to all the <code>rdfs:subClassOf</code> statements
Example	<pre> PREFIX prim: <http://www.ontotext.com/owlim/RDFPriming#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> ASK { rdfs:subClassOf prim:assignWeight "0.5" . }</pre>

Function	Schedule Nodes for Activation
Predicate	http://www.ontotext.com/owlim/RDFPriming#activateNode
Description	Used to schedule the nodes specified as Subject or Object of the statement pattern for activation. Scheduling for activation can also be performed by evaluating an ASK query with variables in the body, in which case the nodes bound to the variables used in the query will be scheduled for activation. The behaviour of such an ASK query is altered, so that all the solutions are exhausted before returning the query result. This could take a long time, since LIMIT and OFFSET are not available in this case. The first example activates two nodes gossip:hasTrack and prel:hasChild and the second example activates many nodes identifying people (and their names) that have an album called "American Life".
Example	<pre> PREFIX prim: <http://www.ontotext.com/owlim/RDFPriming#> PREFIX gossip: <http://www.ontotext.com/rascalli/2008/04/gossipdb.owl#> PREFIX prel: <http://proton.semanticweb.org/2007/10/proton_rel#> ASK { gossip:hasTrack prim:activateNode prel:hasChild }</pre> <pre> PREFIX gossip: <http://www.ontotext.com/rascalli/2008/04/gossipdb.owl#> PREFIX onto: <http://www.ontotext.com#> ASK { ?person gossip:hasAlbum ?album . ?album gossip:name "American Life" . ?person gossip:name ?name }</pre>

The following URI's are used with conjunction with the `<http://www.ontotext.com/owlim/RDFPriming#decayFactor>` predicate to change the parameters of the RDF Priming module. In general, the names of the parameters are Subjects of the statement pattern and the new values are passed as its Object.

Parameter	Activation Threshold
Predicate	http://www.ontotext.com/owlim/RDFPriming#activationThreshold
Description	During activation spreading activations are accumulated in nodes and can grow indefinitely. The <code>activationThreshold</code> allows the user to trim those value to a certain threshold. The default value of this parameter is 1.0, which means that all values bigger than 1.0 are set to 1.0 on every iteration. This parameter is applied on every iteration of the process and guarantees that no activations larger than the parameter value will be encountered.
Example	<pre> PREFIX prim: <http://www.ontotext.com/owlim/RDFPriming#> ASK { prim:activationThreshold prim:setParam "0.9" . }</pre>

Parameter	Decay Factor
Predicate	http://www.ontotext.com/owlim/RDFPriming#decayFactor
Description	Is used during spreading activation to control how much a node's activation level is transferred to nodes that it affects. The following example query sets the new <code>decayFactor</code> to "0.55"
Example	<pre> PREFIX prim: <http://www.ontotext.com/owlim/RDFPriming#> ASK { prim:decayFactor prim:setParam "0.55" . }</pre>

Parameter	Default Activation Value
Predicate	http://www.ontotext.com/owlim/RDFPriming#defaultActivation
Description	Sets the default activation value for all nodes in the repository. If the default activation is not preset then the default activation for all repository nodes is 0. This does not affect the activation origin nodes, whose activation values are set by using http://www.ontotext.com/owlim/RDFPriming#initialActivation

Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:defaultActivation prim:setParam "0.4" . }
----------------	--

Parameter	Default Weight
Predicate	http://www.ontotext.com/owlim/RDFPriming#defaultWeight
Description	Edges in the RDF graph can be given weights that are multiplied by the source node activation in order to compute the activation that is spread across the edge to the destination node (see <code>assignWeight</code>). If the predicate of the edge is not given any specific weight (via <code>assignWeight</code>) then the edge weight is assumed to be 1/3 (one third). This default weight can be changed by using the <code>defaultWeight</code> parameter. Any floating point value in the range 0,1 can be used.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:defaultWeight prim:setParam "0.2" . }

Function	Export Activation Values
Predicate	http://www.ontotext.com/owlim/RDFPriming#exportActivations
Description	Is used to export activation values for a set of nodes. The values are stored in a file identified by the URL given as the Object of the statement pattern. The format of the data in the file is simply one line per URI followed by a tab character and the floating-point value of its activation value.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:exportActivations prim:setParam "file:///D:/work/my_activations.txt" . }

Parameter	Filter Threshold
Predicate	http://www.ontotext.com/owlim/RDFPriming#filterThreshold
Description	Sets the new filter threshold value used to decide when a statement is visible depending on the activation level of its subject, predicate and object.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:filterThreshold prim:setParam "0.50" . }

Parameter	Firing Threshold
Predicate	http://www.ontotext.com/owlim/RDFPriming#firingThreshold
Description	Sets the threshold above which a node will activate its neighbours
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:firingThreshold prim:setParam "0.25" . }

Function	Import Activation Values
Predicate	http://www.ontotext.com/owlim/RDFPriming#importActivations
Description	Is used to import activation values for a set of nodes. The values are loaded from a file identified by the URL given as the Object of the statement pattern. The format of the data in the file is simply one line per URI followed by a tab character and the floating-point value of its activation value.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:importActivations prim:setParam "file:///D:/work/my_activations.txt" . }

Parameter	Initial Activation Value
Predicate	http://www.ontotext.com/owlim/RDFPriming#initialActivation
Description	Sets the initial activation value for each of the nodes from which the activation process starts. The nodes that are scheduled for activation will receive that amount at the beginning of the spreading activation process.

Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:initialActivation prim:setParam "0.66" . }
----------------	---

Parameter	Maximum Nodes Fired Per Cycle
Predicate	http://www.ontotext.com/owlim/RDFPriming#maxNodesFiredPerCycle
Description	Sets the number of nodes that should fire activations during one spreading activation cycle. The default value is 10000.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:maxNodesFiredPerCycle prim:setParam "10000" . }

Parameter	Number of Cycles
Predicate	http://www.ontotext.com/owlim/RDFPriming#cycles
Description	Sets the number of activation spreading cycles to perform when the process is initiated.
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:cycles prim:setParam "4" . }

Parameter	Number of Worker Threads
Predicate	http://www.ontotext.com/owlim/RDFPriming#workerThreads
Description	Sets the number of worker threads that will perform the spreading activation (the default is 2).
Example	PREFIX prim: < http://www.ontotext.com/owlim/RDFPriming# > ASK { prim:workerThreads prim:setParam "4" . }

RDF Priming Example

The following example uses data from DBPEDIA <http://dbpedia.org/About> and was imported into OWLIM-SE with the RDF Priming mode enabled. The management queries are evaluated through the Sesame console application for convenience. The initial step is to evaluate a demo query that retrieves all the instances of the `dbpedia:V8` concept:

```
SELECT *
WHERE { ?x <http://dbpedia.org/property/class> <http://dbpedia.org/resource/V8> . }
```

The above query returns the following results:

```
?x
-----
dbpedia3:Jaguar_AJ-V8_engine
dbpedia3:BMW_M62
dbpedia3:BMW_N62
dbpedia3:Chrysler_Flathead_engine
dbpedia3:Duramax_V8_engine
dbpedia3:Ford_385_engine
dbpedia3:Ford_MEL_engine
dbpedia3:Ford_Power_Stroke_engine
dbpedia3:Ford_Y-block_engine
dbpedia3:Ford_Yamaha_V8_engine
dbpedia3:GM_Premium_V_engine
dbpedia3:Lincoln_Y-block_V8_engine
dbpedia3:Mercedes-Benz_M113_engine
dbpedia3:Nissan_VH_engine
dbpedia3:Nissan_VK_engine
dbpedia3:BMW_N63
dbpedia3:Toyota_UR_engine
dbpedia3:Toyota_UZ_engine
```

As can be seen, the query returns many engines from different manufacturers. The RDF Priming module can be used to reduce the number of results returned by this query by targeting the query to specific parts of the global RDF graph, i.e. the parts of the graph that have been activated.

The following text shows an example of setting up and configuring the RDF Priming module for the purpose of making the example query return a smaller set of more specific results. It is assumed that a SPARQL endpoint is available that is connected to a running repository instance.

Enable the RDF Priming module:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { _:b1 onto:enableSpreading "true" . }
```

Change the default decay factor:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { onto:decayFactor onto:setParam "0.55" . }
```

Change the firing threshold parameter:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { onto:firingThreshold onto:setParam "0.25" . }
```

Change the filter threshold:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { onto:filterThreshold onto:setParam "0.60" . }
```

The initial Activation Level is changed to reflect the specifics of the data set:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { onto:initialActivation onto:setParam "0.66" . }
```

Adjust the Weight factors for a specific predicate so that it activates the relevant sub-set of the RDF graph, in this case the `rdfs:subClassOf` predicate:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
ASK { rdfs:subClassOf onto:assignWeight "0.5" . }
```

The next step alters the Weight Factor of the `rdf:type` predicate so that it does not propagate activations to the classes from the activated instances. This is a useful technique when there are a lot of instances and a very large classification taxonomy which should not be broadly activated (as is the case with the DBpedia dataset).

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ASK { rdf:type onto:assignWeight "0.1" . }
```

If the example query is executed at this stage, it will return no results, because the RDF graph has no activated nodes at all. Therefore the next step is to activate two particular nodes, the Ford Motor Company `dbpedia3:Ford_Motor_Company` and one of the cars they build `dbpedia3:1955_Ford`, which came out of the factory with a very nice V8 engine:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
PREFIX dbpedia3: <http://dbpedia.org/resource/>
ASK { dbpedia3:1955_Ford onto:activateNode dbpedia3:Ford_Motor_Company }
```

Finally, tell the RDF Priming module to spread the activations from these two nodes:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { _:b0 onto:spreadActivation _:b1 . }
```

This will normally take 8-10 seconds after which the example query can be re-evaluated with the following results:


```
?x
-----
dbpedia3:Jaguar_AJ-V8_engine
dbpedia3:BMW_M62
dbpedia3:Ford_385_engine
dbpedia3:Ford_MEL_engine
dbpedia3:Ford_Y-block_engine
```

As can be seen, the result set is smaller and most of the engines retrieved are made by Ford. However, there is an engine made by Jaguar which is most probably there because Ford owned Jaguar for some time in the past, so both manufacturers are somehow related to each other. This might also be the case for the other non-Ford engines returned, since BMW also owned Jaguar for some time. Of course, these remarks are a free interpretation of the results.

Finally, disable the RDF Priming module:

```
PREFIX onto: <http://www.ontotext.com/owlim/RDFPriming#>
ASK { _:b1 onto:enableSpreading "false" . }
```

to return to the normal operating mode.

OWLIM-SE LUBM

This section describes how to set up and run the [Lehigh University Benchmark \(LUBM\)](#) using the scripts and configuration files included with the OWLIM-SE distribution. Running the tests and the benchmarks may require minor modifications to these scripts and configuration files, as discussed in this section.

- [Configuring OWLIM-SE and Sesame](#)
- [Running the Benchmark](#)

Configuring OWLIM-SE and Sesame

Requirements for running the benchmark:

- Java version 1.5 or higher
- Sesame version 2 or higher

Running the benchmark (as well as the getting started application), requires modification of the script `setvars` (respectively `.cmd` and `.sh`) in OWLIM-SE's main folder. The most important setting is the specification of the Java virtual machine through the `JAVA_HOME` environment variable.

The performance of the benchmark can be tuned to the particular hardware on which it is running with modification of the repository specification file named `lubm.ttl`, which can be found in the `lubm` sub-directory of OWLIM-SE's main directory.

Running the Benchmark

To run the Lehigh University Benchmark (LUBM), it is necessary to generate the test file-set beforehand. This can be done using the `lubm-generate` (`.cmd` or `.sh`, respectively for Windows or Linux) script, which is part of the distribution's `lubm` sub-folder. The distribution includes a pre-built library of the [benchmark's source code](#). The pre-built `lubm.jar` library is located in the `ext` distribution folder and also includes the wrapper classes that the benchmark's code-base uses in order to run against a Sesame repository (configured with OWLIM-SE).

The `lubm-generate` script (`.cmd` or `.sh`) accepts a single numeric argument as the target number of universities and creates a sub-folder for the test run with the name `univer` and the number of universities appended, e.g. `univer1000`. This is the folder in which it places the generated OWL files. Once the file-set is generated, the `lubm.config` file should be edited and the appropriate benchmark configuration to be executed should be commented/uncommented. By default, a single University dataset is configured and its configuration section is:

```
# 1-university
[OWLIM_1]
class=owlim.OwlimWrapper
data=./univer1
database=jdbc:ignore
ontology=http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl
```

One should use the `#` symbol at the beginning of a line to comment it. The test should be launched via the `lubm-benchmark` script. Before either generation of datasets or execution of the test, OWLIM-SE should be configured as discussed in properly. To find out more about this benchmark, visit the [LUBM web site](#).

Before either generation of datasets or execution of the test, OWLIM-SE should be configured as discussed in the [installation section](#) by editing `lubm.ttl`, however the file provided is suitable for running the benchmark with small datasets. To find out more about this benchmark, visit the [LUBM web site](#).

OWLIM-SE Performance Tuning

This section is intended for system administrators that are already familiar with OWLIM-SE and the Sesame openRDF infrastructure, who wish to configure their system for optimal performance. Best performance is typically measured by the shortest load time and fastest query answering. Many factors affect the performance of a OWLIM-SE repository in many different ways. This section is an attempt to bring together all factors that affect performance, however it is measured.

- [Memory configuration](#)
 - [Setting the maximum Java heap space](#)
 - [Data structures](#)
 - [Running in a servlet container](#)
- [Delete operations](#)
 - [Algorithm](#)
 - [Problem](#)
 - [Solution](#)
 - [Example](#)
 - [Schema transactions](#)
- [Optional indices](#)
 - [Predicate lists](#)
 - [Context indices](#)
- [Query performance](#)
 - [Query optimisation](#)
 - [Caching literal language tags](#)
 - [Enumerating owl:sameAs](#)
- [Reasoning complexity](#)
 - [Custom rulesets](#)
 - [Long transitive chains](#)
- [Strategy](#)
 - [Dataset loading](#)
 - [Normal operation](#)

Memory configuration

Memory configuration is the single most important factor for optimising the performance of OWLIM-SE. In every respect, the more memory available the better the performance. The only question, is how to divide up the available memory between the various OWLIM-SE data structures in order to achieve the best overall behaviour.

Setting the maximum Java heap space

The maximum amount of heap space used by a Java virtual machine (JVM) is specified using the `-Xmx` virtual machine parameter. The value should be no higher than the amount of free memory available in the target system multiplied by some factor to allow for extra runtime overhead, say approximately ~90%.

For example, if a system has 16GB total RAM and 1GB is used by the operating system, services etc, then ideally the JVM that hosts the application using OWLIM-SE would have a maximum heap size of 15GB (16-1) and would be set using the JVM argument: `-Xmx15g`

Data structures

The heap space available is used by:

- the JVM, the application and OWLIM-SE workspace (byte code, stacks, etc)
- data structures for storing entities affected by specifying `entity-index-size`
- data structures for indexing statements specified using `cache-memory`

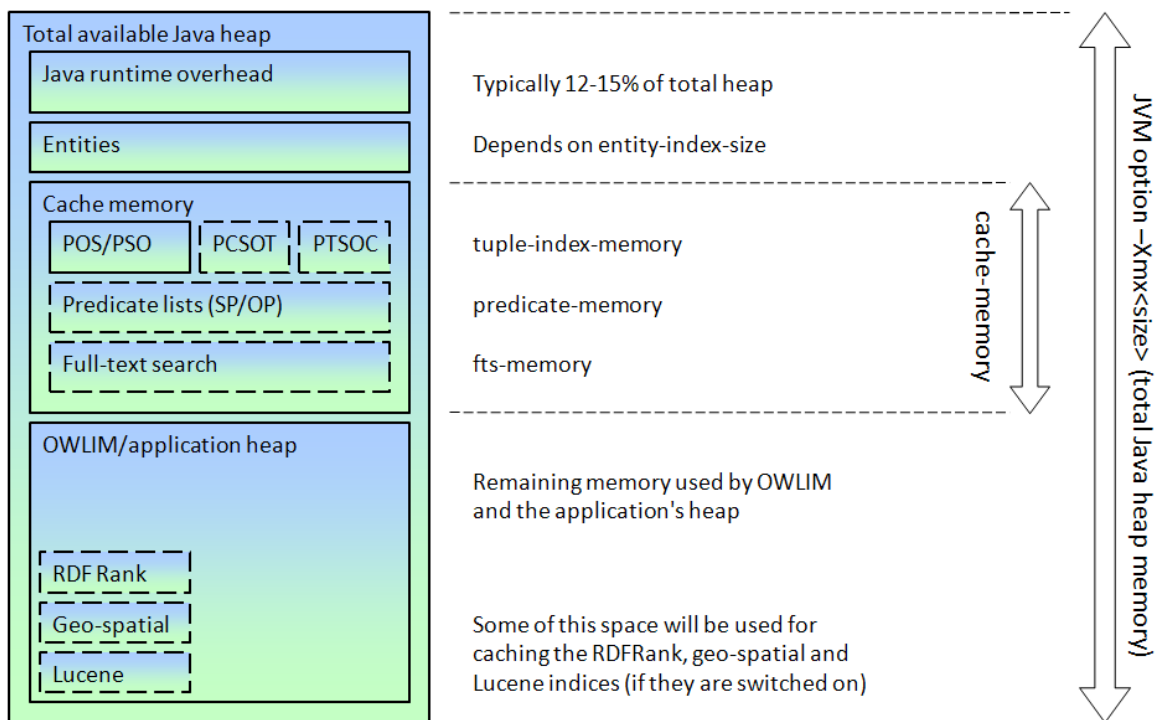
Simplistically, the memory required for storing entities is determined by the number of entities in the dataset, where the memory required is 4 bytes per slot allocated by `entity-index-size` plus 12 bytes for the actual number of entities.

However, the memory required for the indices (cache types) depends on which indices are being used. The `SPO` and `PSO` indices are always used. Optional indices include `predicateLists`, `PCSOT`, `PTSOC` and `FTS` (full-text search) indices.

The memory allocated to these cache types can be calculated automatically by OWLIM-SE, however some of them can be specified in a more fine-grained way. The following configuration parameters are relevant

```
cache-memory = tuple-index-memory + predicate-memory + fts-memory
```

A more complete view of memory use for OWLIM-SE is given here:



During the initial load, some speed up can be achieved by switching journaling off, by setting the `journaling` configuration parameter to `false`. All this does is to stop the transaction log being written to disk (in case of failure), but a noticeable speed up is possible. During normal use, it is recommended to enable journaling for more reliable recovery after an unexpected termination.

Running in a servlet container

If the OWLIM-SE repository is being hosted by the Sesame HTTP servlet then the maximum heap space will apply to the servlet container (tomcat). In which case, allow some more heap memory for the runtime overhead, especially if running at the same time as other servlets. Also, some configuration of the servlet container might improve performance, e.g. increasing the permanent generation, which by default is 64MB. Quadrupling (for tomcat) with `-XX:MaxPermSize=256m` might help. Further information can be found in the tomcat documentation.

Delete operations

OWLIM-SE's inference policy is one of total materialisation, where implicit statements are inferred from explicit statements as soon as they are inserted in to the repository using the specified semantics `ruleset`. This approach has the advantage that query answering can be achieved very quickly, since no inference needs to be done at query time. However, no justification information is stored for inferred statements, therefore deleting a statement would normally require a full re-computation of all inferred statements, which can take a very long time for large datasets. OWLIM-SE uses a special technique for handling the deletion of explicit statements and their inferences called **smooth delete**. This is switched on by default, but can be overridden using the `enableSmoothDelete` configuration parameter set to `false`.

Algorithm

The algorithm used to identify and remove those inferred statements that can no longer be derived using the explicit statements being deleted is as follows:

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

Problem

The difficulty with the current algorithm is that almost all delete operations will follow inference paths that touch schema statements, which then lead to almost all other statements in the repository. This can lead to **smooth delete** taking a very long time indeed.

Solution

What can stop the algorithm touching too many (possibly all) statements, however, is that the algorithm will not go further if a visited statement is marked read-only. Since a read-only statement cannot be deleted, there is no reason to find what statements are inferred from it (such inferred statements might still get deleted, but they will be found by following other inference paths).

Statements are marked as read-only if they occur in `ruleset` files (standard or custom) or are loaded at initialisation time via the `imports` configuration parameter.

Therefore, when using **smooth delete**, it is recommended to load all ontology/schema/vocabulary statements using the `imports` configuration parameter.

Example

Consider the following statements:

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

Data:
<wayne_rooney> <foaf:name> "Davenport"^^xsd:string .
<Reviewer40476> <rdf:type> <MyClass> .
<Reviewer40478> <rdf:type> <MyClass> .
<Reviewer40480> <rdf:type> <MyClass> .
<Reviewer40481> <rdf:type> <MyClass> .
```

When using the `owl-horst` rule set the removal of the statement:

```
<wayne_rooney> <foaf:name> "Davenport"
```

will cause the following sequence of events:

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

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

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

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

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

In large datasets there are typically millions of statements [`X rdf:type owl:Thing`], and they will all be visited by the algorithm. The [`X rdf:type owl:Thing`] statements are not the only problematic statements that will be considered for removal. Every class that has millions of instances will lead to similar behaviour.

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

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

Schema transactions

As mentioned above, ontologies and schemas imported at initialisation time using the 'imports' configuration parameter are flagged as read-only. However, there are times when it is necessary to change a schema and this can be done inside a 'system transaction'. The user instructs OWLIM that the transaction is a system transaction by including a dummy statement with the special `schemaTransaction` predicate, i.e.

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

This statement is not inserted in to the database, rather it serves as a flag that tells OWLIM that it can ignore the read-only flag for imported statements.

Optional indices

Predicate lists

Predicate lists are two indices (`SP` and `OP`) that can improve performance in two separate situations:

- Loading/querying datasets that have a large number of predicates
- Executing queries or retrieving statements that use a wildcard in the predicate position, for example using the statement pattern:
`dbpedia:Human ?predicate dbpedia:Land`

As a rough guideline, a dataset with more than about 1000 predicates will benefit from using these indices for both loading and query answering. Predicate list indices are not enabled by default, but can be switched on using the `enablePredicateList` configuration parameter.

Context indices

Two further indices can also be used for providing better performance when executing queries that use context and/or triples set ids. (Triples set ids are not exposed through the Sesame interface, so can be ignored).

The `PCSOT` index can be enabled using the `build-pcsot` configuration parameter. This index will improve performance when using statement patterns where the context and subject are bound, e.g.

```
?s skos:broader ?o onto:dataset1
```

The `PTSOC` index can be enabled with the `build-ptsoc` configuration parameter. This index will improve performance when using statement patterns where the predicate and subject are bound, e.g.

```
dbpedia:Human rdfs:subClassOf ?superclass ?context
```

Query performance

Query optimisation

OWLIM-SE uses a number of query optimisation techniques by default. These can be disabled by using the `enable-optimization` configuration parameter set to `false`, however there is rarely any need to do this.

Caching literal language tags

This optimization applies when the repository contains a large number of literals with language tags and it is necessary to execute queries that filter based on language, e.g. using the following SPARQL query construct:

```
FILTER ( lang(?name) = "ES" )
```

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

Enumerating `owl:sameAs`

The presence of many `owl:sameAs` statements – such as when using several LOD datasets and link sets – causes an explosion in the number of inferred statements. For a simple example, if A is a city in country X, and B and C are alternative names for A, and Y and Z are alternative names for X, then the inference engine should infer: B in X, C in X, B in Y, C in Y, B in Z, C in Z also.

As described in the OWLIM-SE user guide, OWLIM-SE avoids the inferred statement explosion caused by having many `owl:sameAs` statements by grouping equivalent URIs in to a single master node and using this for inference and statement retrieval. This is in effect a kind of backward chaining that allows all the sound and complete statements to be computed at query time.

This optimisation can save a large amount of space for two reasons:

1. A single node is used for all N URIs in an equivalence class, which avoids storing N `owl:sameAs` statements;
2. If there are N equivalent URIs in one equivalence class then the reasoning engine should infer that all URIs in this equivalence class are the equivalent to each other (and themselves), i.e. another N^2 `owl:sameAs` statements can be avoided.

During query answering, all members of each equivalence appearing in a query are substituted to generate sound and complete query

results. However, even though the mechanism to store equivalences is standard and cannot be switched off, it is possible to prevent the enumeration of equivalence classes during query answering. When using a dataset with many `owl:sameAs` statements, turning off the enumeration can dramatically reduce the number of *duplicated* query results, where a single URI from each equivalence class is chosen to be representative.

To turn off the enumeration of equivalent URIs, a special pseudo-graph name can be used:

FROM/FROM NAMED <<http://www.ontotext.com/disable-SameAs>>

Two different versions of a query are shown below with and without this special graph name. The queries are executed against the [factforge](#) combined dataset:

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

Gives results:

```
dbpedia:Air_strip
http://sw.cyc.com/concept/Mx4ruQS1AL_QQdeZXf-MIWWdng
umbel-sc:CommercialAirport
opencyc:Mx4ruQS1AL_QQdeZXf-MIWWdng
dbpedia:Jetport
dbpedia:Airstrips
dbpedia:Airport
dbpedia:Airporgt
fb:guid.9202a8c04000641f800000000004ae12
opencyc-en:CommercialAirport
```

Whereas the same query with the pseudo-graph that prevents equivalence class enumeration:

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

Gives results:

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

Reasoning complexity

The complexity of the rule set has a large effect on loading performance and on the overall size of the repository after loading. The complexity of the standard rule sets increases as follows:

- none (lowest complexity, best performance)
- rdfs
- rdfs-optimized
- owl-horst-optimized
- owl-horst
- owl-max-optimized
- owl-max
- owl2-ql-optimized
- owl2-ql
- owl2-rl-optimized
- owl2-rl (highest complexity, worst performance)

It should be noted that all rules affect the loading speed, even if they never actually infer any new statements. This is because as new statements are added, they are pushed in to every rule to see if anything is inferred. Often this can result in many joins being computed even though the rule never 'fires'.

Custom rulesets

If better load performance is required and it is known that the dataset does not contain anything that will apply to certain rules then they can be omitted from the ruleset. To do this, copy the appropriate '.pie' file included in the distribution and remove the unused rules. Then

set the ruleset configuration parameter to the full pathname to this custom rule set.

If custom rules are being used to specify semantics not covered by the included standard rulesets, then some care must be taken for the following reasons:

- Recursive rules can lead to an explosion in the number of inferred statements
- Rules with unbound variables in the head cause new blank nodes to be created – the inferred statements can never be retracted and can cause other rules to fire

Long transitive chains

SwiftOWLIM version 2.9 contained a special optimisation that prevents the materialisation of inferred statements as the result of transitive chains. Instead, these inferences were computed during query answering. However, such an optimisation is NOT available in OWLIM-SE due to the nature of the indexing structures. Therefore, OWLIM-SE will attempt to materialise all inferred statements at load time. When a transitive chain is long then this can cause a very large number of inferences to be made. For example, for a chain of N `rdfs:subClassOf` relationships, OWLIM-SE will infer (and materialise) a further $(N^2-N)/2$ statements. If the relationship is also symmetric, e.g. in a family ontology with a predicate such as `relatedTo`, then there will be N^2-N inferred statements.

Administrators should therefore take great care when managing datasets that have long chains of transitive relationships. If performance becomes a problem then it may be necessary to:

1. Modify the schema, either by removing the symmetry of certain transitive relationships or chaining the transitive nature of certain properties altogether
2. Reducing the complexity of inference by choosing a less sophisticated ruleset

Strategy

The life-cycle of a repository instance typically starts with the initial loading of datasets followed by the processing of queries and updates. The loading of a large dataset can take a long time - 12 hours for a billion statements with inference is not unusual. Therefore, it is often useful to use a different configuration during loading than during normal use. Furthermore, if a dataset is frequently loaded, because it changes gradually over time, then the loading configuration can be evolved as the administrator gets more familiar with the behaviour of OWLIM-SE with this dataset. Many properties of the dataset only become apparent after the initial load (such as the number of unique entities) and this information can be used to optimise the loading step the next time round or to improve the normal use configuration.

Dataset loading

A typical initialisation life-cycle would be like this:

1. Configure a repository for best loading performance with many parameters estimated
2. Load data
3. Examine dataset properties
4. Refine loading configuration
5. Reload data and measure improvement

Unless the repository needs to answer queries during the initialisation phase, the repository can be configured with the minimum number of options and indices, with a large portion of the available heap space given over to the cache memory:

```
enablePredicateList = false (unless the dataset has a large number of predicates)
build-pcsot = false
build-ptsoc = false
in-memory-literal-properties = false
journaling = false
cache-memory = approximately 50% of total heap space (-Xmx value)
```

Normal operation

The optional indices can be built at a later time when the repository is used for query answering. The details of all optional indices, caches and optimisations have been covered previously in this document. Some experimentation is required using typical query patterns from the user environment.

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

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

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

The file `entities.index` is used to look up entries in the file `entities.hash` and its size is equal to the value of the `entity-index-size` parameter multiplied by 4. Therefore the `entity-index-size` parameter has less to do with efficient use of memory and more to do with the performance of entity indexing and lookup. The larger this value, the less collisions occur in the `entities.hash` table. A reasonable size for this parameter is at least half the number of unique entities. However, the size of this data structure is never changed once the repository is created, so this knowledge can only be used to adjust this value for the next clean load of the dataset with a new (empty) repository.

The following parameters can be adjusted:

parameter	Comment
entity-index-size	set to a large enough value as described above
enablePredicateList	can speed up queries (and loading)
build-pcsot	
build-ptsoc	
in-memory-literal-properties	
journaling	Should be set to true
cache-memory	
tuple-index-memory	
predicate-memory	if predicate lists are enabled
fts-memory	if using Node Search

Don't forget that:

```
cache-memory = tuple-index-memory + predicate-memory + fts-memory
```

If any one of these is missed out, it will be calculated. If two or more are unspecified then the remaining cache memory is divided evenly between them.

Furthermore, the inference semantics can be adjusted by choosing a different rule set. However, this will require a reload of the whole repository, otherwise some inferences can remain when they should not.

OWLIM-SE Performance Highlights

A detailed comparison of the performance of the major semantic repositories in existence today can be found in [20] section 'State of the Art in Semantic Repositories'.

The latest OWLIM-SE benchmark results for a range of tests (LUBM, OUBM, BSBM, etc) are maintained here:
<http://www.ontotext.com/owlim/benchmarking/>

OWLIM-SE References

1. About WordNet. Home page. <http://wordnet.princeton.edu/>
2. Aduna b. v. *User Guide of Sesame*. <http://www.openrdf.org/doc/sesame/users/index.html>
3. Beckett, D. (editor). (2004). *RDF/XML Syntax Specification (Revised)*. W3C Recommendation 10 Feb. 2004. <http://www.w3.org/TR/rdf-syntax-grammar/>
4. Brickley, D.; Guha, R.V. *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C (10 Feb 2004) <http://www.w3.org/TR/rdf-schema/>
5. Broekstra, J. (2005). *Storage, Querying and Inferencing for Semantic Web Languages*. Ph.D. Thesis, Vrije Universiteit Amsterdam, SIKS Dissertation Series No. 2005-09, ISBN 90 9019 2360. <http://www.cs.vu.nl/~jbroeks/#pub>
6. Carroll, J. J.; De Roo, Jos. (2004). *OWL Web Ontology Language: Test Cases*. W3C Recommendation 10 Feb. 2004. <http://www.w3.org/TR/owl-test/>
7. Dean, M; Schreiber, G. – editors; Bechhofer, S; van Harmelen, F; Hendler, J; Horrocks, I.; McGuinness, D. L; Patel-Schneider, P. F.; Stein, L. A. (2004). *OWL Web Ontology Language Reference*. W3C Recommendation, 10 Feb. 2004. <http://www.w3.org/TR/owl-ref/>
8. Grosz, B; Horrocks, I; Volz, R; Decker, St. (2003). *Description Logic Programs: Combining Logic Programs with Description Logic*. In Proc. of WWW2003, Budapest, May 2003.
9. Guo, Y; Pan, Z; and Heflin, J. (2004). *An Evaluation of Knowledge Base Systems for Large OWL Datasets*. Journal of Web Semantics, 3(2), 2005, pp158-182. <http://www.websemanticsjournal.org/ps/pub/2005-16>
10. Hayes, P. (2004). *RDF Semantics*. W3C Recommendation 10 Feb. 2004. <http://www.w3.org/TR/2004/REC-rdf-mt-20040210/>
11. Horrocks, I., Patel-Schneider, P. F., Bechhofer, S., Tsarkov, D. OWL Rules: A Proposal and Prototype Implementation. [Journal of Web Semantics 3 \(2005\), pp. 23-40.](http://www.websemanticsjournal.org/ps/pub/2005-16)
12. Janik, M., Kochut, K. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In Proc. of ISWC 2005, Galway, Ireland, November 6-10, 2005. LNCS 3729, pp. 431-445.
13. Kiryakov, A. *Validation Goals and Metrics for the LarKC Platform*. LarKC project deliverable D5.5.1, 2009
14. Kiryakov, A; Ognyanov, D; Manov, D. (2005). *OWLIM – a Pragmatic Semantic Repository for OWL*. In Proc. of International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2005), WISE 2005, 20 Nov, New York City, USA.
15. Klyne, G; Carrol, J. J; (eds). (2004). *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation 10 Feb. 2004. <http://www.w3.org/TR/rdf-concepts/>
16. Ma, L; Yang, Y; Qiu, Z; Xie, G; Pan, Y. *Towards A Complete OWL Ontology Benchmark*. In Proc. of the 3rd European Semantic Web Conference (ESWC 2006). Budva (Montenegro).
17. Motik, B., Sattler, U., Studer, R. *Query Answering for OWL-DL with Rules*. [Journal of Web Semantics 3 \(2005\), pp. 41-60.](http://www.websemanticsjournal.org/ps/pub/2005-16)
18. ter Horst, H. J. *Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity*. In Proc. of ISWC 2005, Galway, Ireland, November 6-10, 2005. LNCS 3729, pp. 668-684.
19. TRREE – Triple Reasoning and Rule Entailment Engine. Home page. <http://ontotext.com/trree/>
20. Rule Interchange Format (RIF) W3C Working Group, *OWL2 RL in RIF*, <http://www.w3.org/2005/rules/wiki/OWLRL>
21. Weithöner, T., Liebig, T., Luther, M., Böhm, S. (2006). *What's Wrong with OWL Benchmarks?* SSWS2006
22. William F. Clocksin, Christopher S. Mellish: *Programming in Prolog: Using the ISO Standard*. Springer, 5th ed., 2003, ISBN 978-3540006787
23. Hitzler, Pascal; Krötzsch, Markus; Parsia, Bijan; Patel-Schneider, Peter F.; Rudolph, Sebastian (27 October 2009). "OWL 2 Web Ontology Language Primer". OWL 2 Web Ontology Language. World Wide Web Consortium. <http://www.w3.org/TR/2009/REC-owl2-primer-20091027/>
24. Motik, Boris; Cuenca Grau, Bernardo; Horrocks, Ian; Wu, Zhe; Fokoue, Achille; Lutz, Carsten (27 October 2009) "OWL 2 Web Ontology Language Profiles". OWL 2 Web Ontology Language. World Wide Web Consortium. <http://www.w3.org/TR/owl2-profiles/>