Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema

Jeen Broekstra[1], Arjohn Kampman[1], and Frank van Harmelen[2]

[1] Aidministrator Nederland b.v., Amersfoort, The Netherlands
{jeen.broekstra, arjohn.kampman}@aidministrator.nl
[2] Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands
frank.van.harmelen@cs.vu.nl

Abstract. RDF and RDF Schema are two W3C standards aimed at enriching the Web with machine-processable semantic data.
We have developed Sesame, an architecture for efficient storage and expressive querying of large quantities of metadata in RDF and RDF Schema. Sesame's design and implementation are independent from any specific storage device. Thus, Sesame can be deployed on top of a variety of storage devices, such as relational databases, triple stores, or object-oriented databases, without having to change the query engine or other functional modules. Sesame offers support for concurrency control, independent export of RDF and RDFS information and a query engine for RQL, a query language for RDF that offers native support for RDF Schema semantics. We present an overview of Sesame as a generic architecture, as well as its implementation and our first experiences with this implementation.

1 Introduction

The Resource Description Framework (RDF) [14] is a W3C Recommendation for the formulation of metadata on the World Wide Web. RDF Schema [4] (RDFS) extends this standard with the means to specify domain vocabulary and object structures. These techniques will enable the enrichment of the Web with machine-processable semantics, thus giving rise to what has been dubbed the Semantic Web.

We have developed Sesame, an architecture for storage and quering of RDF and RDFS information. Sesame is being developed by Aidministrator Nederland b.v.[3] as part of the European IST project On-To-Knowledge[4] [9]. Sesame allows persistent storage of RDF data and schema information, and provides access methods to that information through export and querying modules. It features ways of caching information and offers support for concurrency control.

This paper is organized as follows. In section 2 we give a short introduction to RDF and RDFS. Readers who are already familiar with these languages can

[3] See http://www.aidministrator.nl/
[4] On-To-Knowledge (IST-1999-10132). See http://www.ontoknowledge.org/

safely skip this section. In section 3 we discuss why a query language specifically tailored to RDF and RDFS is needed, over and above existing query languages such as XQuery. In section 4 we look at Sesame's modular architecture in some detail. In section 5 we give an overview of the SAIL API and a brief comparison to other RDF API approaches. Section 6 discusses our experiences with Sesame until now, and section 7 looks into possible future developments. Finally we provide our conclusions in section 8.

2 RDF and RDFS

RDF is a W3C recommendation that was originally designed to standardize the definition and use of metadata-descriptions of Web-based resources. However, RDF is equally well suited for representing arbitrary data, be they metadata or not.

The basic building block in RDF is an subject-predicate-object triple, commonly written as $P(S, O)$. That is, a subject S has an predicate (or property) P with value O. Another way to think of this relationship is as a labeled edge between two nodes: $[S] - P \rightarrow [O]$.

This notation is useful because RDF allows subjects and objects to be interchanged. Thus, any object from one triple can play the role of a subject in another triple, which amounts to chaining two labeled edges in a graphic representation. The graph in figure 1 for example, expresses three statements.

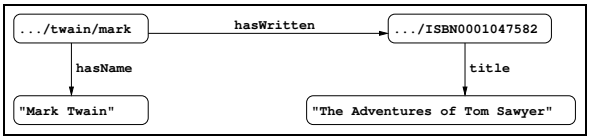

Fig. 1: An example RDF graph.

RDF also allows a form of reification in which any RDF statement itself can be the subject or object of a triple. This means graphs can be nested as well as chained. On the Web this allows us, for example, to express doubt or support for statements created by other people.

The RDF Model and Syntax specification also proposes an XML syntax for RDF data models. One possible serialization of the above relations in this syntax looks like this:

```
<rdf:Description rdf:about="http://www.famouswriters.org/twain/mark">
    <s:hasName>Mark Twain</s:hasName>
    <s:hasWritten rdf:resource="http://www.books.org/ISBN0001047582"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.books.org/ISBN0001047582">
    <s:title>The Adventures of Tom Sawyer</s:title>
    <rdf:type rdf:resource="http://www.description.org/schema#Book"/>
</rdf:Description>
```

Since the proposed XML syntax allows many alternative ways of writing down information, the above XML syntax is just one of many possibilities of writing down an RDF model in XML.

It is important to note that RDF is designed to provide a basic subject-predicate-object model for Web-data. Other than this intended semantics – de-

scribed only informally in the standard – RDF makes no data modeling commitments. In particular, no reserved terms are defined for further data modeling. As with XML, the RDF data model provides no mechanisms for declaring vocabulary that is to be used.

RDF Schema is a mechanism that lets developers define a particular vocabulary for RDF data (such as the predicate hasWritten) and specify the kinds of objects to which predicates can be applied (such as the class Writer). RDFS does this by pre-specifying some terminology, such as Class, subClassOf and Property, which can then be used in application-specific schemata. RDFS expressions are also valid RDF expressions – in fact, the only difference with 'normal' RDF expressions is that in RDFS an agreement is made on the *semantics* of certain terms and thus on the *interpretation* of certain statements. For example, the subClassOf property allows the developer to specify the hierarchical organization of classes. Objects can be declared to be instances of these classes using the type property. Constraints on the use of properties can be specified using domain and range constructs.

Above the dotted line in figure 2, we see an example RDF schema that defines vocabulary for the RDF example we saw earlier: Book, Writer and FamousWriter are introduced as classes, and hasWritten is introduced as a property. A specific instance is described below the dotted line, in terms of this vocabulary.

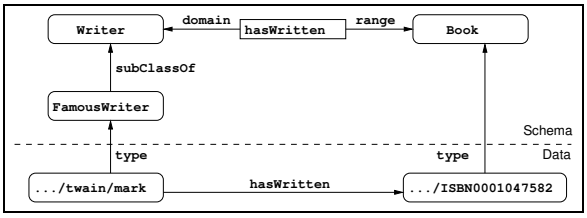

Fig. 2: An example RDF Schema.

3 The need for an RDFS Query Language

RDF documents and RDF schemata can be considered at three different levels of abstraction:

1. at the *syntactic level* they are XML documents.[5]
2. at the *structure level* they consist of a set of triples.
3. at the *semantic level* they constitute one or more graphs with partially predefined semantics.

We can query these documents at each of these three levels. We will briefly consider the pros and cons of doing so for each level in the next sections. This will lead us to conclude that RDF(S) documents should really be queried at the semantic level. We will also briefly discuss RQL, a language for querying RDF(S) documents at the semantic level, which has been implemented in the Sesame architecture.

[5] Actually, this is not necessarily true; non-XML syntaxes for RDF exist, but XML is the most widely used syntax for RDF.

3.1 Querying at the syntactic level

As we have seen in section 2, any RDF model (and therefore any RDF schema) can be written down in XML notation. It would therefore seem reasonable to assume that we can query RDF using an XML query language (for example, XQuery [8]).

However, this approach disregards the fact that RDF is not just an XML notation, but has its own data model that is very different from the XML tree structure. Relationships in the RDF data model that are not apparent from the XML tree structure become very hard to query.

As an example, consider again the XML description of the RDF model in figure 1. In an XML query language such as XQuery [8], expressions to traverse the data structure are tailored towards traversing a node-labeled tree. However, the RDF data model is a graph, not a tree, and moreover, both its edges (properties) and its nodes (subjects/objects) are labeled. In querying at the syntax level, this is literally left as an excercise for the query builder: one cannot query the relation between the resource signifying 'Mark Twain' and the resource signifying 'The Adventures of Tom Sawyer' without knowledge of the syntax that was used to encode the RDF data in XML.

Ideally, we would want to formulate a query like "Give me all the relationships that exist between Mark Twain and The Adventures of Tom Sawyer". However, using only the XML syntax, we are stuck with formulating an awkward query like "Give me all the elements nested in a Description element with an about attribute with value 'http://www.famouswriters.org/twain/mark', of which the value of its resource attribute occurs elsewhere as the about attribute value of a Description element that has a nested element title with the value 'The Adventures of Tom Sawyer'."

Not only is this approach inconvenient, it also disregards the fact that the XML syntax for RDF is not unique: the same RDF graph can be serialized in XML in a variety of ways. This means that one query will never be guaranteed to retrieve all the answers from an RDF model.

3.2 Querying at the structure level

When we abstract from the syntax, any RDF document represents a set of triples, each triple representing a statement of the form Subject-Predicate-Object. A number of query languages have been proposed and implemented that regard RDF documents as such a set of triples, and that allow to query such a triple set in various ways.

Look again at the example from 2. An RDF query language such as, for example, Squish [15] would allow us to query which resources are known to be of type FamousWriter:

```
SELECT ?x
FROM   somesource
WHERE  (rdf::type ?x FamousWriter)
```

The clear advantage of such a query is that it directly addresses the RDF data model, and that it is therefore independent of the specific syntax that has been chosen to represent the data.

However, a disadvantage of any query language at this level is that it interprets *any* RDF model only as a set of triples, including those elements which have been given a special semantics in RDFS. For example, since .../twain/mark is of type FamousWriter, and since FamousWriter is a subclass of Writer, .../twain/mark is also of type Writer, by virtue of the intended RDFS semantics of type and subClassOf. However, there is no triple that explicitly asserts this fact. As a result, the query

```
SELECT ?x
FROM   somesource
WHERE  (rdf::type ?x Writer)
```

will fail because the query only looks for explicit triples in the store, whereas the triple `(/twain/mark, type, Writer)` is not explicitly present in the store, but is implied by the semantics of RDFS.

3.3 Querying at the semantic level

What is clearly required is the means to query at the *semantic level*, that is, querying the full knowledge that a RDFS description entails and not just the explicitly asserted statements.

There are at least two options to achieve this goal:

1. Compute and store the closure of the given graph as a basis for querying.
2. Let a query processor infer new statements as needed per query.

While the choice of an RDF query language is, in principle, independent of the choice made in this respect, the fact remains that most RDF query languages have been designed to query a simple triple base, and have no specific functionality or semantics to discriminate between schema and data information.

RQL [13,1] is a proposal for a declarative query language that does expliticly capture these semantics in the language design itself. The language has been initially developed by the Institute of Computer Science at FORTH[6], in Heraklion, Greece, in the context of the European IST project MESMUSES[7]. We will briefly describe the language here; for a detailed description of the language see [13,5].

RQL adopts the syntax of OQL [7], and like OQL, the language is defined by means of a set of core queries, a set of basic filters, and a way to build new queries through functional composition and iterators.

The core queries are the basic building blocks of RQL, which give access to the RDFS specific contents of an RDF triple store. RQL allows queries such as Class (retrieving all classes), Property (retrieving all properties) or Writer (returning all

[6] See `http://www.ics.forth.gr`
[7] See `http://cweb.inria.fr/Projects/Mesmuses/`

instances of the class with name Writer). This last query returns of course also all instances of subclasses of Writer, since these are also instances of the class Writer, by virtue of the semantics of RDFS. Notice that in RQL, these semantics are defined in the query language itself: the formal query language definition makes a commitment to interpret the semantics of RDFS. This is notably different from an approach like Squish, where the designer/implementer is at liberty to interpret the RDFS entailment using one of the options mentioned earlier, or not at all.

For composing more complex queries, RQL has a select-from-where construction. In the from-clause of such a query, we can specify a *path expression*. These allow us to match patterns along entire paths in RDF/RDFS graphs. For example, the query

```
select Y, $Y
from FamousWriter{X}.hasWritten{Y : $Y}
```

returns all things written by famous writers, and the type of that thing, effectively doing pattern-matching along a path in the graph of figure 2. Notice that RQL path expressions explicitly enable free mixing of data and schema information.

4 Sesame's Architecture

Sesame is an architecture that allows persistent storage of RDF data and schema information and subsequent querying of that information. In section 4.1, we present an overview of Sesame's architecture. In the sections following that, we look in more detail at several components.

4.1 Overview

An overview of Sesame's architecture is shown in Figure 3. In this section we will give a brief overview of the main components.

For persistent storage of RDF data, Sesame needs a scalable repository. Naturally, a Data Base Management System (DBMS) comes to mind, as these have been used for decades for storing large quantities of data. In these decades, a large number of DBMS's have been developed, each having their own strengths and weaknesses, targeted platforms, and API's. Also, for each of these DBMS's, the RDF data can be stored in numerous ways.

As we would like to keep Sesame DBMS-independent and it is impossible to know which way of storing the data is best fitted for which DBMS or which application domain, all DBMS-specific code is concentrated in a single architectural layer of Sesame: the *Storage And Inference Layer* (SAIL).

This SAIL is an application programming interface (API) that offers RDF-specific methods to its clients and translates these methods to calls to its specific DBMS. An important advantage of the introduction of such a separate layer is that it makes it possible to implement Sesame on top of a wide variety of

repositories without changing any of Sesame's other components. Section 5 looks at the API in more detail.

Sesame's functional modules are clients of the SAIL API. Currently, there are three such modules: The *RQL query engine*, the *RDF admin module* and the *RDF export module*. Each of these modules is described in more detail in section 4.2.

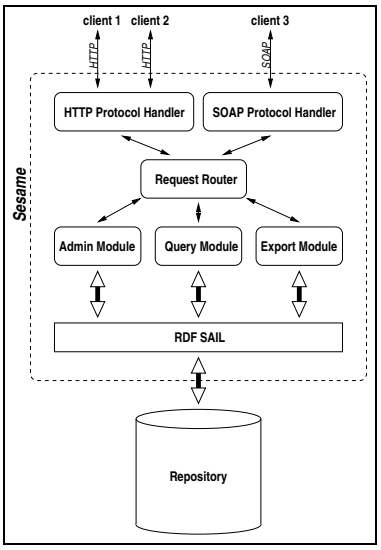

Fig. 3: Sesame's architecture.

Depending on the environment in which it is deployed, different ways to communicate with the Sesame modules may be desirable. For example, communication over HTTP may be preferable in a Web context, but in other contexts protocols such as Remote Method Invocation (RMI) or the Simple Object Access Protocol (SOAP) [3] may be more suited.

In order to allow maximal flexibility, the actual handling of these protocols has been placed outside the scope of the functional modules. Instead, protocol handlers are provided as intermediaries between the modules and their clients, each handling a specific protocol.

The introduction of the SAIL and the protocol handlers makes Sesame into a generic architecture for RDFS storage and querying, rather than just a particular implementation of such a system.

Adding additional protocol handlers makes it easy to connect Sesame to different operating environments. The construction of concrete SAIL implementations will be discussed in section 5.

Sesame's architecture has been designed with extensibility and adaptability in mind. The possibility to use other kinds of repositories has been mentioned before. Adding additional modules or protocol handlers is also possible.

4.2 Sesame's functional modules

The RQL query module As we have seen, one of the three modules currently implemented in Sesame is an RQL query engine.

In Sesame, a version of RQL was implemented that is slightly different from the language proposed by [13]. The Sesame version of RQL features better compliance to W3C specifications, including support for optional domain- and range restrictions as well as multiple domain- and range restrictions. It does, however, not feature support for datatyping as proposed in the original language proposal. See [5] for details.

The Query Module follows the path depicted in figure 4 when handling a query. After parsing the query and building a query tree model for it, this model is fed to the query optimizer which transforms the query model into an equivalent model that will evaluate more efficiently. These optimizations mainly consist

of a set of heuristics for query subclause move-around. Notice that these pre-evaluation optimizations are not dependent on either domain or storage method.

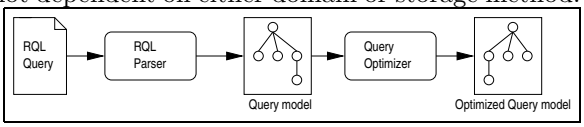

Fig. 4: Query parsing and optimization model.

The optimized model of the query is subsequently evaluated in a streaming fashion, following the tree structure into which the query has been broken down. Each object represents a basic unit in the original query and evaluates itself, fetching data from the SAIL where needed. The main advantage of this approach is that results can be returned in a streaming fashion, instead of having to build up the entire result set in memory first.

In Sesame, RQL queries are translated (via the object model) into a set of calls to the SAIL. This approach means that the main bulk of the actual evaluation of the RQL query is done in the RQL query engine itself.

For example, when a query contains a semi-join operation over two subqueries, each of the subqueries is evaluated, and the semi-join operation is then executed by the query engine on the results.

Another approach would be to directly translate as much of the RQL query as possible to a query specific for the underlying repository. An advantage of this approach is that, when using a DBMS, we would get all its sophisticated query evaluation and optimization mechanisms for free. However, a large disadvantage is that the implementation of the query engine is directly dependent on the repository being used, and the architecture would lose the ability to easily switch between repositories.

This design decision is one of the major differences between Sesame and the RDF Suite implementation of RQL by ICS-FORTH (see [1]). The RDF Suite implementation relies on the underlying DBMS for query optimization. However, this dependency means that RDF Suite cannot as easily be transported to run on top of another storage engine.

A natural consequence of our choice to evaluate queries in the SAIL is that we need to devise several optimization techniques in the engine and the SAIL API implementation, since we cannot rely on any given DBMS to do this for us.

The admin module In order to be able to insert RDF data and schema information into a repository, Sesame provides an admin module. The current implementation is rather simple and offers two main functions:

1. incrementally adding RDF data/schema information;
2. clearing a repository.

Partial delete (on a per-statement basis) functionality is not yet available in the current admin module, but support for this feature is under development.

The admin module retrieves its information from an RDF(S) source (usually an online RDF(S) document in XML-serialized form) and parses it using a streaming RDF parser (currently, we use the ARP RDF parser that is part of the Jena toolkit [6]). The parser delivers the information to the admin module

on a per-statement basis: `(Subject, Predicate, Object)`. The admin subsequently tries to assert this statement into the repository by communicating with the SAIL and reports back any errors or warnings that might have occurred.

The current implementation makes no explicit use of the transaction-functionality of SAIL yet, but we expect to implement this in the near future.

The RDF export module The RDF Export Module is a very simple module. This module is able to export the contents of a repository formatted in XML-serialized RDF. The idea behind this module is that it supplies a basis for using Sesame in combination with other RDF tools, as all RDF tools will at least be able to read this format.

Some tools, like ontology editors, only need the schema part of the data. On the other hand, tools that don't support RDFS semantics will probably only need the non-schema part of the data. For these reasons, the RDF Export Module is able to selectively export the schema, the data, or both.

5 The SAIL API

The SAIL API is a set of Java interfaces that has been specifically designed for storage and retrieval of RDFS-based information. The main design principles of SAIL are that the API should:

- define a basic interface for storing RDF and RDFS in, and retrieving and deleting RDF and RDFS from (persistent) repositories.
- abstract from the actual storage mechanism; it should be applicable to RDBMSs, file systems, or in-memory storage, for example.
- be usable on low end hardware like PDAs, but also offer enough freedom for optimizations to handle huge amounts of data efficiently on e.g. enterprise level database clusters.
- be extendable to other RDF-based languages like DAML+OIL [11].

Other proposals for RDF APIs are currently under development. The most prominent of these are the Jena toolkit [6] and the Redland Application Framework [2]. SAIL shares many characteristics with both approaches.

An important difference between these two proposals and SAIL, is that the SAIL API specifically deals with RDFS on the retrieval side: it offers methods for querying class and property subsumption, and domain and range restrictions. In contrast, both Jena and Redland focus exclusively on the RDF triple set, leaving interpretation of these triples as an excercise to the user. In SAIL, these RDFS inferencing tasks are handled internally. The main reason for this is that there is a strong relationship between the efficiency of the inferencing and the actual storage model being used. Since any particulary SAIL implementation has a complete understanding of the storage model (e.g. the database schema in the case of an RDBMS), this knowledge can be exploited to infer, for example, class subsumption more efficiently.

Another difference between SAIL and other RDF APIs is that SAIL is considerably more lightweight: only four basic interfaces are pre-defined, offering basic storage and retrieval functionality and transaction support, but not much beyond that. We feel that in some applications such minimality may be preferable to an API that has more features, but is also more complex to understand and implement.

The current Sesame system offers several implementations of the SAIL API. The most important of these is the SQL92SAIL, which is a generic implementation for SQL92 [12]. The aim is to be able to connect to any RDBMS while having to re-implement as little as possible. In the SQL92SAIL, only the definitions of the datatypes (which are not part of the SQL92 standard) have to be changed when switching to a different database platform. The SQL92SAIL features an inferencing module for RDFS, based on the RDFS entailment rules as specified in the RDF Model Theory [10]. This inferencing module computes the schema closure of the RDFS being uploaded, and asserts these implicates of the schema as derived statements. For example, whenever a statement of the form `(foo, rdfs:domain, bar)` is encountered, the inferencing module asserts that `(foo, rdf:type, property)` is an implied statement.

The SQL92SAIL has been tested in use with several DBMSs, including PostgreSQL[8] and MySQL[9] (see also section 6).

An important feature of the SAIL (or indeed of any API) is that it is possible to put one on top of the other. The SAIL at the top can perform some action when the modules make calls to it, and then forward these calls to the SAIL beneath it. This process continues until one of the SAILs finally handles the actual retrieval request, propagating the result back up again.

We implemented a SAIL that caches all schema data in a dedicated data structure in main memory. This schema data is often very limited in size and is requested very frequently. At the same time, the schema data is the most difficult to query from a DBMS because of the transitivity of the subClassOf and subPropertyOf properties. This schema-caching SAIL can be placed on top of arbitrary other SAILs, handling all calls concerning schema data. The rest of the calls are forwarded to the underlying SAIL.

Another important task that can be handled by a SAIL is concurrency handling. Since any given RQL query is broken down into several operations on the SAIL level, it is important to preserve repository consistency over multiple operations. We implemented a SAIL that selectively blocks and releases read and write access to repositories, on a first come first serve basis. This setup allows us to support concurrency control for any type of repository.

6 Experiences

Our implementation of Sesame can be found at `http://sesame.aidministrator.nl/`, and is freely available for non-commercial use. The implementation follows

[8] See `http://www.postgresql.org/`
[9] See `http://www.mysql.com/`

the generic architecture described in this paper, using the following concrete implementation choices for the modules:

- We use both PostgreSQL and MySQL as database platforms. The reason we are running two platforms simultaneously is mainly a development choice: we wish to compare real-life performance.
- platforms. We have various repository setups running, combining different stacks of SAILs (including the SQL92SAIL, the PostgreSQL SAIL, the MySQL SAIL, and a schema cache and a concurrency handler) on top of each repository.
- A protocol handler is realised using HTTP.
- The admin module uses the ARP RDF parser.

In this section, we briefly report on our experiences with various aspects of this implementation.

6.1 RDFS in practice

While developing Sesame, many unclarities in the RDFS specification were uncovered. One of the reasons for this is that RDFS is defined in natural language: no formal description of its semantics is given. As a result of this, the RDFS specification even contains some inconsistencies.

In an attempt to solve these unclarities, the RDF Core Working Group has been chartered to revise the RDF and RDFS specifications. One of the results is a formal Model Theory for RDF [10], which specifies model and schema semantics more precisely and includes a formal procedure for computing the closure of a schema.

As mentioned in section 5, the SQL92SAIL features an inferencing module that follows the procedure described in the model theory. Our experiences are that a naive implementation of this formal procedure is painfully slow, but with relative ease it can be optimized to perform quite satisfactory. Improving this performance even further is currently work in progress.

6.2 PostgreSQL and SAIL

In our first test setup for Sesame we used PostgreSQL. PostgreSQL is a freely available (open source) object-relational DBMS that supports many features that normally can only be found in commercial DBMS implementations (see `http://www.postgresql.org`).

One of the main reasons for choosing PostgreSQL is that it is an object-relational DBMS, meaning that it supports subtable relations between its tables. As these subtable relations are also transitive, we used these to model the class and property subsumption relations of RDFS.

The SAIL that is used in this setup therefore is specifically tailored towards PostgreSQL's support for subtables (which is not a standard SQL feature). It uses a dynamic database schema that was inspired by the schema shown in [13].

New tables are added to the database whenever a new class or property is added to the repository. If a class is a subclass of other classes, the table created for it will also be a subtable of the tables for the superclasses. Likewise for properties being subproperties of other properties. Instances of classes and properties are inserted as values into the appropriate tables. Figure 5 gives an impression of the contents of a database containing the data from figure 2.

The actual schema involves one more table called resources. This table contains all resources and literal values, mapped to a unique ID. These IDs are used in the tables shown in the figure to refer to the resources and literal values. The resources table is used to minimize the size of the database. It ensures that resources and literal values, which can be quite long, only occur once in the database, saving potentially large amounts of memory.

In the test setup, several optimizations in the SAIL implementation were made, such as selective caching of namespaces and frequently requested resources to avoid repetitive table lookups.

Our experiences with this database schema on PostgreSQL were not completely satisfactory. Data insertion is not as fast as we would like. Especially incremental uploads of schema data can be very slow, since table creation is very expensive in PostgreSQL. Even worse, when adding a new subClassOf relation between two existing classes, the complete class hierarchy starting from the subclass needs to broken down and rebuilt again because subtable relations can not be added to an existing table; the subtable relations have to be specified when a table is created. Once created, the subtable relations are fixed. Another disadvantage of the subtable-approach is that cycles in the class hierarchy can not be modeled properly in this fashion.

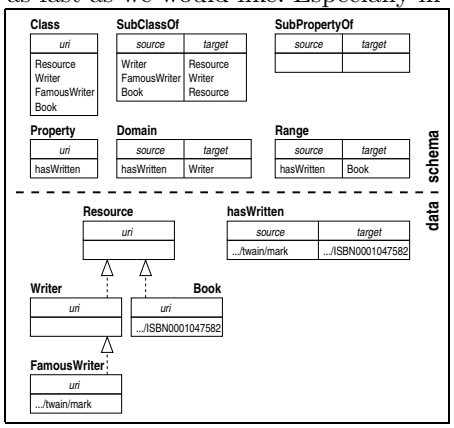

Fig. 5: Impression of the object-relational schema used with PostgreSQL

In a new setup, we used the SQL92SAIL to connect to PostgreSQL. The current version of this SAIL implementation takes a radically different approach: all RDF statements are inserted into a single table with three columns: `Subject, Predicate, Object`. While we have yet to perform structured testing and analysis with this approach, it seems to perform significantly better, especially in scenarios where the RDFS changes often.

For querying purposes, the original PostgreSQL SAIL performed quite satisfactory, especially when combined with a Schema-caching SAIL stacked on top (see section 5). We have yet to perform structured testing on querying with the new SQL92SAIL, but initial results show that it performs somewhat slower than the earlier PostgreSQL SAIL, which is to be expected.

6.3 MySQL

In initial tests with MySQL, we implemented a SAIL with a strictly relational database schema (see figure 6).

As can be seen, a number of dependencies arise due to the storage of Schema information in separate tables. In order to keep overhead to a minimum, every resource and literal is encoding using an integer value (the `id` field), to enable faster lookups. To encode whether a particular statement was explicitly asserted or derived from the schema information, an extra column `is_derived` is added where appropriate.

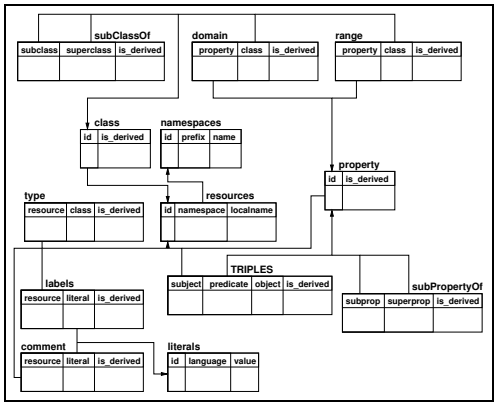

Fig. 6: Impression of the relational schema used with MySQL.

The main difference between this schema and the schema used in the PosgreSQL setup (see figure 5) is that in this setup, the database schema does not change when the RDFS changes. In application scenarios where the RDFS (the ontology) is unstable, this is an advantage because typically adding new tables to a database requires more time and resources than simply inserting a row in an existing table.
Like in the PostgreSQL SAIL, selective caching of namespaces and other optimization techniques were implemented in this setup. Overall, this approach performed significantly better in our test scenarios, especially on uploading.

7 Future work

7.1 Transaction rollback support

While the SAIL API has support for transactions, it currently has no transaction rollback feature. Transaction rollbacks, especially in the case of uploading information, are crucial if we wish to guarantee database consistency. In the case of RDF uploads, transaction rollbacks can be supported at two levels:

- a single upload of a set of RDF statements can be seen as a single transaction, or alternatively, a single upload can be "chunked" into smaller sets to support partial rollback when an error occurs during the upload session.
- a single RDFS statement assertion can been seen as a transaction in which several tables in the database need to be updated. From the user point of view, the schema assertion is atomic ("A is a class"), but from the repository point of view, it may consist of several table updates, for instance, in the schema presented in figure 5, a new table would have to be created, and new rows would have to be inserted into the "Resources" and the "Classes" table.

Both levels of transaction rollback support may help ensure database consistency. Together with the concurrency support already present in the Sesame system, this will help move Sesame towards becoming an ACID[10] compliant storage system (note that this can only be guaranteed if the platform used for storage supports it).

7.2 Versioning support

The current version of Sesame has no support for versioning. However, concrete plans for implementing a per-statement form of versioning exist. This basic type of versioning will enable more elaborate versioning schemes.

7.3 Adding and extending functional modules

Sesame currently features three functional modules. We plan to extend the functionality of these modules, as well as add new modules.

In the current admin module implementation, only incremental upload of RDF statements is supported. We plan to implement more advanced update support, most importantly support for deleting individual triples from the repository. A prototype implementation of this new feature already exists but has to be tested and extended further.

Plans for new modules include a graphical visualization component and query engines for different query languages (for example, Squish).

7.4 DAML+OIL support

As mentioned in section 5, the RDF SAIL API has been designed to allow extension of the functionality, for example to include support for DAML+OIL.

In the current implementation, this support is not present however. We plan to implement at least partial support for DAML+OIL storage and inferencing.

8 Conclusions

In this paper we have presented Sesame, a generic architecture for storing and querying both RDF and RDFS information. Sesame is an important step beyond the currently available storage and query devices for RDF, since it is the first publicly available implementation of a query language that is aware of the RDFS semantics.

An important feature of the Sesame architecture is its abstraction from the details of any particular repository used for the actual storage. This makes it possible to port Sesame to a large variety of different repositories, including relational databases, RDF triple stores, and even remote storage services on the Web.

[10] *Atomicity, Concurrency, Isolation, Durability.* These four properties of a transaction ensure database robustness over aborted or (partially) failed transactions.

Sesame itself is a server-based application, and can therefore be used as a remote service for storing and querying data on the Semantic Web. As with the storage layer, Sesame abstracts from any particular communication protocol, so that Sesame can easily be connected to different clients by writing different protocol handlers.

We have constructed several concrete implementations of the generic architecture, using PostgreSQL and MySQL as repositories and using HTTP as communication protocol handlers.

Important next steps to expand Sesame towards a full fledged storage and querying service for the Semantic Web include implementing transaction rollback support, versioning, extension from RDFS to DAML+OIL and implementations for different repositories. This last feature especially will be greatly facilitated by the fact that the current SAIL implementation is a generic SQL92 implementation, rather than specific for a particular DBMS.

References

1. Sofia Alexaki, Vassilis Christophides, Greg Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The RDFSuite: Managing Voluminous RDF Description Bases. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece, 2000. See `http://www.ics.forth.gr/proj/isst/RDF/RSSDB/rdfsuite.pdf`.
2. David Beckett. The Design and Implementation of the Redland RDF Application Framework. In *Proceedings of Semantic Web Workshop of the 10th International World Wide Web Conference*, Hong-Kong, China, May 2001.
3. Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3c note, World Wide Web Consortium, May 2000. See `http://www.w3.org/TR/SOAP/`.
4. D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium, March 2000. See `http://www.w3.org/TR/2000/CR-rdf-schema-20000327`.
5. Jeen Broekstra and Arjohn Kampman. Query Language Definition. On-To-Knowledge (IST-1999-10132) Deliverable 9, Aidministrator Nederland b.v., April 2001. See `http://www.ontoknowledge.org/`.
6. Jeremy Carrol and Brian McBride. The Jena Semantic Web Toolkit. Public api, HP-Labs, Bristol, 2001. See `http://www.hpl.hp.com/semweb/jena-top.html`.
7. R.G.G. Cattel, Douglas Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
8. Don Chamberlin, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery: A Query Language for XML. Working draft, World Wide Web Consortium, February 2001. See `http://www.w3.org/TR/xquery/`.
9. Dieter Fensel, Frank van Harmelen, Michel Klein, Hans Akkermans, Jeen Broekstra, Christiaan Fluit, Jos van der Meer, Hans-Peter Schnurr, Rudi Studer, John Hughes, Uwe Krohn, John Davies, Robert Engels, Bernt Bremdal, Fredrik Ygge, Thorsten Lau, Bernd Novotny, Ulrich Reimer, and Ian Horrocks. On-To-Knowledge: Ontology-based Tools for Knowledge Management. In *Proceedings of the eBusiness and eWork 2000 (EMMSEC 2000) Conference*, Madrid, Spain, October 18–20, 2000.

10. Patrick Hayes. RDF Model Theory. Working draft, World Wide Web Consortium, September 2001. See http://www.w3.org/TR/rdf-mt/.
11. Ian Horrocks, Frank van Harmelen, Peter Patel-Schneider, Tim Berners-Lee, Dan Brickley, Dan Connoly, Mike Dean, Stefan Decker, Dieter Fensel, Pat Hayes, Jeff Heflin, Jim Hendler, Ora Lassila, Deborah McGuinness, and Lynn Andrea Stein. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html, March 2001.
12. ISO. Information Technology-Database Language SQL. Standard No. ISO/IEC 9075:1999, International Organization for Standardization (ISO), 1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.).
13. Gregory Karvounarakis, Vassilis Christophides, Dimitris Plexousakis, and Sofia Alexaki. Querying community web portals. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece, 2000. See http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.pdf.
14. O. Lassila and R. R. Swick. Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium, February 1999. See http://www.w3.org/TR/REC-rdf-syntax/.
15. Libby Miller. RDF Squish query language and Java implementation. Public draft, Institute for Learning and Research Technology, 2001. See http://ilrt.org/discovery/2001/02/squish/.