Key Words: Category: H.2.8 Database Applications,
Category: H.3.3 Information Search and Retrieval,
Category: I.2.0 Artificial Intelligence, General

3store: Efficient Bulk RDF Storage

Stephen Harris
(University of Southampton, United Kingdom
swh@ecs.soton.ac.uk)

Nicholas Gibbins
(University of Southampton, United Kingdom
nmg@ecs.soton.ac.uk)

Abstract: The development and deployment of practical Semantic Web applications requires technologies for the storage and retrieval of RDF data that are robust and scalable. In this paper, we describe the 3store RDF storage and query engine developed within the Advanced Knowledge Technologies project, and discuss the design rationale and optimisations behind it which enable the efficient handling of large RDF knowledge bases.

1 Introduction

The `hyphen.info` RDF dataset, which describes computer science research in the UK, was constructed by members of the Advanced Knowledge Technologies (AKT) [2] project as a testbed for Semantic Web research, and has been used as a foundation on which to build Semantic Web applications and investigate tools. This dataset describes the people, projects, publications and institutions involved in computer science research. It is gathered from a number of heterogeneous sources on a variety of schedules from daily to monthly, and is expressed in RDF form using the AKT Reference Ontology [3].

We initially developed a simple RDF storage layer (3store 1.x) with limited inferential capability in order to hold and query this data. This prototype consisted of a PHP module that implemented the OKBC "get" query methods [10] on top of a MySQL database engine; the majority of the inference performed by the system was handled by the SQL engine, but the implementation in PHP still imposed a significant overhead and made it hard to provide native interfaces for other languages. Although it fulfilled our immediate requirements, our experiences with it have highlighted a number of shortcomings and limitations and have lead to a new set of requirements for a more capable RDF bulk storage engine for the AKT project.

This paper describes the design and development of the 3store engine. It focuses in turn on the necessary requirements for large-scale Semantic Web applications, on the approaches adopted by existing systems of equivalent capabilities, and on the technical aspects of 3store's design and operation. In particular, we describe those aspects relating to the efficient processing of queries and generation of RDF(S) entailments.

2 Requirements

2.1 Scale

The key requirement for an RDF storage solution for the AKT project are that it must be able to store the `hyphen.info` data, the ontologies that make up the AKT Reference Ontology and other RDF data and schemata that will be needed at a later date.

The `hyphen.info` dataset consists of around 5 million RDF triples when serialised, and is expected to grow to several tens of millions if fully populated with information from all the computer science institutions in the UK. The AKT Reference Ontology consists of around 200 classes and 150 properties, and we expect to include several more separate ontologies and sub-ontologies of a similar size.

From this, the base scale requirements were decided to be the ability to handle at least 20 million triples and 5000 classes and properties. The knowledge base must also be able to import and replace RDF data in sufficient time to keep pace with the nightly re-gathered data.

2.2 Interfaces

The previous (prototype) version of our 3store software supported an RDF-based dialect of OKBC which used HTTP as its transport layer. This was intended as a lightweight interface by which RDF-aware clients could invoke the knowledge base as a web service; the OKBC API calls were PHP scripts which, when invoked with appropriate parameters, returned an XML or RDF document containing the results of the call. The OKBC-HTTP interface was used in a number of our existing applications, so the maintenance of this interface was a requirement to ensure backwards compatibility with previous versions, as well as an opportunity to reimplement it in a more efficient manner as an Apache module that wrapped a C library with a direct SQL connection.

In addition to this OKBC API, we felt it appropriate to implement a more natural RDF query interface, based on the RDQL[18] query language. Our existing applications had made heavy use of the stored procedure capability of the OKBC API, indicating that the simple API calls were not expressive enough to support the development of more sophisticated applications. This RDQL interface, which is described further in Section 4.5, provides an HTTP interface that returns the results in an XML format, and a database-style C API that queries the knowledge base directly.

2.3 Inferential capability

The model theory [16] that forms part of the RDF and RDF Schema specifications (which are, at the time of writing, in the latter stages of standardisation) describes a number of inferences or entailments that may be generated by matching rules in an RDF document. These entailments are labelled rdf1 and rdfs2–rdfs10, and are reproduced for convenience in Table 1. In this table, aaa stands for any uriref, and uuu for any uriref or bNode. xxx, yyy and zzz can represent any node in the graph. The column labelled 'Eval' indicates whether an entailment is evaluated at assertion time (eagerly, denoted E), or lazily at query time (denoted L).

Our existing applications, based on the previous version of 3store with its OKBC interface, required entailments rdfs5a–6 and rdfs7a–9, which are respectively the transitive closures of the rdfs:subPropertyOf and rdfs:subClassOf properties. In addition, it was

Label	Rule	Entailment	Eval
rdf1	xxx aaa yyy	aaa rdf:type rdf:Property	E
rdfs2	xxx aaa yyy aaa rdfs:domain zzz	xxx rdf:type zzz	E
rdfs3	xxx aaa uuu aaa rdfs:range zzz	uuu rdf:type zzz	E
rdfs4a	xxx aaa yyy	xxx rdf:type rdfs:Resource	E
rdfs4b	xxx aaa uuu	uuu rdf:type rdfs:Resource	E
rdfs5a	aaa rdfs:subPropertyOf bbb bbb rdfs:subPropertyOf ccc	aaa rdfs:subPropertyOf ccc	E
rdfs5b	xxx rdf:type rdf:property	xxx rdfs:subPropertyOf xxx	E
rdfs6	xxx aaa yyy aaa rdfs:subPropertyOf bbb	xxx bbb yyy	L
rdfs7a	xxx rdf:type rdfs:Class	xxx rdfs:subClassOf rdfs:Resource	E
rdfs7b	xxx rdf:type rdfs:Class	xxx rdfs:subClassOf xxx	E
rdfs8	xxx rdfs:subClassOf yyy yyy rdfs:subClassOf zzz	xxx rdfs:subClassOf zzz	E
rdfs9	xxx rdfs:subClassOf yyy aaa rdf:type xxx	aaa rdf:type yyy	L
rdfs10	xxx rdf:type rdfs:ContainerMembershipProperty	xxx rdfs:subPropertyOf rdfs:member	E

Table 1: RDF(S) entailments

expected that we would also make use of entailments rdfs2–4 and rdf1, so these were added as required entailments.

At present we deal only with RDF(S) entailments, but we anticipate introducing support for a limited set of features from the OWL Web Ontology Language [21], as described further in Section 7.

2.4 Efficiency

We consider the time efficiency of 3store to consist of two separate facets: the efficiency of evaluating queries, and the efficiency of asserting new knowledge.

Many of the applications under development in AKT require interactive-level performance while evaluating queries containing significant numbers of constraints. For example, when formulated in RDQL the CS AKTiveSpace [26] user interface uses queries with between four and twelve triple patterns in the `WHERE` clause, returning a few hundred result rows.

These applications are commonly based around a Web browser interface; the expectation of most users is that such an interface should be no less responsive than simple Web browsing, so response time for the queries used must be kept to the order of a few milliseconds on available hardware, in order to maintain the responsiveness of the interfaces.

An orthogonal concern is that of the time taken to assert new knowledge. The knowledge sources that the AKT project uses as a testbed for its research applications are gathered on a variety of schedules ranging from daily to monthly. Maintaining the integrity of the data while it is being reasserted is an important concern, and for this reason the time during which the knowledge base is potentially inconsistent or incomplete should be kept to a minimum. Given the minimum gathering schedule, it should be possible to assert the entire AKT RDF knowledge base in a few hours and replace significant portions of it in a reasonable time for an overnight batch process.

3 Related Work

As part of the decision to update our existing RDF storage technology, we evaluated the suitability of the existing RDF engines for our uses. These engines ranged considerably in their capabilities (some offered complete RDF inference, some offered none), but we concluded that none of the existing technologies offered the balance of performance and inferential ability that we require for our applications. We present a summary of our experiences in the remainder of this section.

3.1 Jena

Jena [19] is a Java toolkit for manipulating RDF models which has been developed by Hewlett-Packard Labs. It has excellent support for RDQL queries, but does not provide an OKBC interface (however, given the level of RDQL support provided, the addition of an OKBC compatibility layer would be straightforward for the portion of the OKBC API that was implemented in the previous version of 3store).

We attempted to import one of hyphen's larger RDF files (comprising around 5% of the data) into Jena, using its default MySQL back-end, but after 24 hours it had not completed the import (the preliminary indications were that it was repeatedly refreshing its database indexes during the import). From this experience, we consider Jena to be unsuitable for storing the volume of data that we require.

3.2 Sesame

Sesame [9] is an architecture for inferencing and querying of RDF and RDF Schema which supports RDQL queries among others (though not OKBC) and produces all the RDF and RDFS entailments.

Sesame performs its query operations in memory and Beckett, 2002 [5] suggests that the query response time for DBMS backed, moderately sized knowledge bases will be in excesses of 60ms, which is not fast enough for our interactive user interfaces.

Sesame was not tested for its import performance.

3.3 Parka

Parka [27] is an inferencing database written over a custom relational database back-end; the ParkaSW version has been modified to support RDF. Query execution is handled by the relational engine, as in 3store, though the table layout is significantly different, so comparing the translation algorithms is beyond the scope of this document.

According to its documentation [20], Parka has an upper limit on its knowledge base size of around 2.5 million triples, which appears to be due to the structure of the relational indexes, and which makes it inappropriate for the scale of data with which we are working.

3.4 Redland

The Redland suite appears to be capable of storage of large RDF graphs, but currently has no graph matching query facility, so is unsuitable for our purposes. Adding a query mechanism to Redland was considered, but because it does not use a DBMS back-end it would have been considerably more effort to implement that than to port the existing 3store code to another environment, given our previous experience.

However we did decide to use the RDF parser from Redland, Raptor [6], which provides a C API for extracting the triples from RDF/XML and RDF/Ntriples documents. It abstracts the complexities of the RDF syntax, and removes the implementation burden from our core goal of building a scalable, persistent RDF knowledge base.

3.5 TAP

TAP KB is an RDBMS backed RDF knowledge base which has appeared since the start of the development of 3store version 2. It provides an Apache module, TAPache, which allows remote access to the stored RDF, much like 3store. However, TAPache provides no graph matching query interface such as RDQL, and provides only the direct triple matching method GetData [28], which is much like the low level librdfsql access methods used internally by 3store, and a C API which provides limited support for RDFS entailments (chiefly rdfs8, subclass transitivity).

4 The Design of the 3store RDF knowledge base

Our evaluation of the existing, publicly available RDF storage technologies suggested that none could efficiently handle the data with which we are working. For this reason, we felt it necessary to develop an RDF store based on the design principles of 3store version 1, which from our previous experiences we knew could scale to handle a few million triples efficiently.

4.1 Platform

For scalability and portability reasons the software is developed for POSIX compliant UNIX environments in ANSI C. C is efficient and can easily be interfaced to other languages such as PHP, Perl, Python, C++ and Java.

The licence chosen was the GNU General Public Licence [14], a Free Software licence, that allows free distribution under certain conditions. Although a significant portion of 3store consists of libraries, the decision was made to licence it all under the General GPL, rather than the Lesser GPL. This should not cause problems to proprietary software as there are non-library interfaces to the query engine.

For ease of development reasons the back-end storage is provided by an SQL engine. After some basic benchmarking MySQL was chosen as the sole target back-end; picking a single back-end rather than a variety allows tighter integration and more optimisation work to be done, without extensive libraries of code for each target back-end. For example the the RDF import stage makes use of MySQL-specific table locking semantics which speed up bulk imports.

MySQL is open source, and portable to many POSIX operating systems, so this was felt to be a good base for persistent storage and querying, and unlikely to be a limitation on portability.

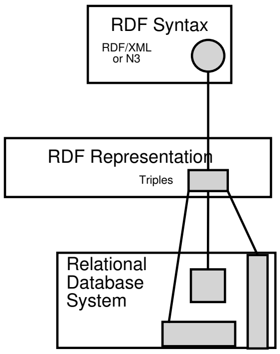


Figure 1: 3store's RDF adaptation of the 3 layer model, after Norrie et al. [23]

Although the MySQL optimiser is not particularly sophisticated it has already be shown in 3store version 1 to be adequate for optimising RDF graph matching queries, which in the database schema used are transformed to an SQL query expression containing a large number of simple joins across a small number of tables, as shown in Figure 2.

4.2 3 layer model

The existing RDF engines that support medium sized knowledge bases and database back-ends (such as Jena and Sesame, discussed in Sections 3.1 and 3.2 respectively) mostly evaluate queries in the RDF part of the engine, rather than in the underlying database. We believe that this misses an opportunity to let the database perform much of the work using its built-in query optimiser (based on its understanding of the indexes). Our experiences from the previous version of 3store indicated that this can quantitatively reduce query execution times.

The 3-layer optimisation model of HYWIBAS[23] and SOPHIA[1] is used to perform multi-level optimisations, and enable efficient RDBMS storage. Like SOPHIA, and unlike HYWIBAS, 3store uses a unified storage mechanism for both classes and instances, in part due to the underlying RDF syntax of RDF Schema. 3store's layers can be characterised as RDF syntax, RDF Representation and Relational Database System, which are comparable to HYWIBAS' Knowledge Base System, Object Data Management System and Relational Database System.

The goal of this work has been to design and implement a system for scalable storage of RDF data, rather than to implement an RDF parser (a non-trivial task given the intricacies of the RDF/XML syntax). We use an off-the-shelf RDF parser, namely the Free Software Raptor toolkit [6], to parse our RDF/XML sources, and pass the resulting triples to 3store's rdfsql library to assert them into the knowledge base. This library translates the triples into SQL `INSERT` statements and passes them to the RDBMS. The 3store code also provides a module for the Apache web server that links against librdfsql and libokbc (the core of 3store) to provide RDQL and OKBC queries over HTTP.

triples

model	subject	predicate	object	literal	inferred
int64	int64	int64	int64	boolean	boolean

models

hash	model
int64	text

resources

hash	uri
int64	text

literals

hash	literal
int64	text

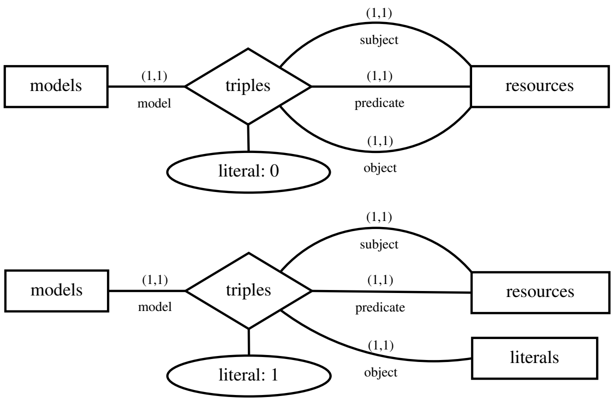


Figure 2: Database schema and ER diagram showing table relationships

4.3 Database structure

The arrangement of tables in the SQL database schema used in the current version of 3store is shown in Figure 2. This schema is normalised using a hash of the resource URIs and literal values as a foreign key in the table which represents the triples in order to reduce the storage requirements for this table, and to ensure that the records in the table are all of the same size (an optimisation which benefits the MySQL database engine). Resources and literals are stored in separate tables with a hash of their values used as the primary key. The same hashing function is used for both resources and for literals, so the triples table contains a flag to indicate whether the object of the triple is a literal or a resource.

The prototype version of 3store used approximately the same table structure, but encoded the namespace URI in addition to the full resource URI for each resource, with an additional $(hash, uri)$ namespaces table. This allowed for more efficient generation of streamed RDF/XML files, as the set of namespaces used in a given subset of triples could be quickly and efficiently found ahead of time, allowing the entity and namespace declarations to be generated before the XML describing the triples. However, this adds an additional burden at assertion time because the namespace prefix and localname components must be identified for each asserted resource, and we found that this was more of a significant issue than the streamed exporting of RDF/XML.

4.4 Hashing

The behaviour of the foreign keys which link the tables in the database schema is informed by the hashing function which is used to generate those keys. We chose to use a 64 bit hash, because this is the largest integer format that commodity servers can handle natively. Picking an integer size that can be handled natively and is supported by the database allows for more efficient indexing and therefore joining of tables, which is an important factor in the query performance of the system.

The "Birthday Paradox"[1] gives the probability of a hash collision occurring anywhere in a knowledge base of 500 million resources and literals to be around $1 : 10^{-10}$ when using a 64 bit hash, and we detect and report any key collisions at assertion time to guard against this (admittedly slim) possibility. Although the same hashing function is used for both resources and literals (which would lead to a hash collision in the event of a URI-valued literal having the same value as a resource), collisions between resource hashes and literal hashes are avoided in the current schema by the addition of a flag which indicates the type of the object of a triple.

The total number of resources and literals depends on the graph structure. The maximum number of resources and literals, which occurs when the RDF graph is an unconnected graph where no two triples share resource or literal values, is three times as many resources as triples (when all triples have resources as objects) or twice as many resources as triples and as many literals as triples (when all triples have literal-valued objects) triples in resources or a number of literals equal to the number of triples, although the typical numbers of resources and literals are much lower.

A simple benchmark was conducted to select between two of the most common 64 bit or greater hashing functions, CRC-64 and and the MD5 digest checksum. While MD5 produces a 128 bit output, only 64 are used. The portion of the MD5 hash used is unimportant as the blocks are independent and unbiased [24]. The benchmark consisted of hashing 8 randomly selected URIs in sequence 100,000 times and recording the number of CPU cycles consumed by each hash function. The test were performed on an Intel Pentium III, the code was generated by gcc 3.2 and the hashing implementations used were taken from GNU libc.

On average the CRC-64 function consumed 1062 cycles per hash and MD5 consumed 1091. However the MD5 function is a universal hash (a hash function is universal if it has the property that the probability of a collision between keys x and y when $x \neq y$ is the same for all x and y [11]), which has advantages when confronted with adversarial environments by making it hard to find colliding keys [12]. However it has a large enough output space for this application, unlike more efficient universal hashes, such as UMAC [7]. This is obviously not a conclusive study and more work should be done to establish if there are other more suitable universal hashing functions.

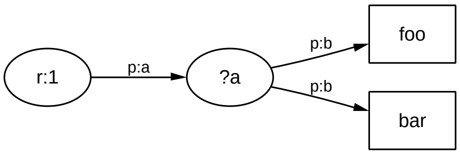
4.5 Query execution

As mentioned in Section 4.2, the 3store RDQL engine transforms an RDQL query into an SQL query over the underlying RDBMS representation of the RDF data. The RDQL query language [18] was defined by the Jena project and has an SQL-like syntax. RDQL is not the only RDF query language; although it is one of the more common, there are other graph matching query languages with comparable syntaxes and capabilities, such as RQL [17] and SquishQL [22]. The example query expressions in this section will be given in RDQL, a sample of which follows:

```
SELECT ?a
WHERE  (<r:1>, <p:a>, ?a),
       (?a, <p:b>, "foo"),
       (?a, <p:b>, "bar")
```

[1] The Birthday Paradox gives the hash collision probability as $p = 1 - (1 - 2^{-s})^n$ where s is the size of the hash (in bits) and n is the number of items (resources in this case). However this assumes a perfectly even hashing function.

The triple syntax in RDQL is (*subject, predicate, object*), so the example above maps to the following RDF graph:



In the example, `?a` represents a free variable and the return values will be all the values of `?a` for which there exists a sub-graph of the knowledge base's RDF graph which matches the pattern graph in the query. RDQL also allows an optional constraints clause, as follows:

```
SELECT ?a
WHERE  (?a, <p:a>, ?c)
AND    ?c != "foo"
```

This translates to "find all the ?a's that have a p:a that is not equal to foo". The RDQL grammar specifies the $==$, $!=$, $<=$, $<=$, $>$ and $<$ operators and allows implementation specific functions. The literals in 3store are treated implicitly as strings, limited type support has been provided by explicit relational operators applied to the lexical representation of the value. This is inefficient and faster mechanisms need to be explored.

Transforming the RDQL triple expressions into relational calculus is relatively straightforward. Each triple pattern in the expression is assigned an existentially quantified variable, denoted in the example by t_n. A constraint is then added to the scope of the existential function for each node appearing in more than one triple. Constraints are also added for every object's resource/literal constraint, where known. Finally, a free tuple variable is included for any requested attributes, such as `?a` in the example. Using this method, the first RDQL example above can be transformed into the following relation calculus expression:

$$
\begin{array}{l}
\{r.uri \mid \text{resource}(r) \wedge \\
\qquad (\exists t_1)(\exists t_2)(\exists t_3)(\text{triple}(t_1) \ \wedge\ \text{triple}(t_2) \ \wedge\ \text{triple}(t_3) \wedge \\
\qquad\qquad\qquad r.hash = t_1.object \wedge \\
\qquad\qquad\qquad t_1.object = t_2.subject \ \wedge\ t_1.object = t_3.subject \wedge \\
\qquad\qquad\qquad t_1.predicate = \text{hash(p:a)} \ \wedge \\
\qquad\qquad\qquad \text{not(t1.literal)} \wedge \\
\qquad\qquad\qquad t_2.object = \text{hash(foo)} \ \wedge\ t_2.literal \wedge \\
\qquad\qquad\qquad t_3.object = \text{hash(bar)} \ \wedge\ t_3.literal \ )\}
\end{array}
$$

The resource/literal type of any free variables in the query can often be determined by inspection, such as when it appears as the subject or predicate of a triple expression, when it is explicitly bound to a URI constant in the constraints section of the RDQL query, or when it appears as the value of a predicate whose range is known. In these situations the graph can easily be transformed to a relational calculus expression. However, this is not the case when the type of the requested attribute is unknown, as in the following example.

```
SELECT ?a WHERE (<r:1>, <p:a>, ?a)
```

In this case, the relational expression becomes more complex:

$$
\begin{aligned}
\{r.uri, l.literal \mid (\exists t_1)(\ & \text{resource}(r) \ \wedge \ \text{triple}(t_1) \wedge \\
& t_1.subject = \text{hash(r:1)} \wedge \\
& t_1.predicate = \text{hash(p:a)} \wedge \\
& t_1.object = r.hash \ \wedge \ \text{not(t1.literal)} \ \wedge \\
& l.literal = \text{null}) \vee \\
(\exists t_2)(\ & \text{literal}(l) \ \wedge \ \text{triple}(t_2) \wedge \\
& t_2.subject = \text{hash(r:1)} \wedge \\
& t_2.predicate = \text{hash(p:a)} \wedge \\
& t_2.object = l.hash \ \wedge \ t_2.literal \ \wedge \\
& r.uri = \text{null})\}
\end{aligned}
$$

We characterised this in SQL as a left join, which has some performance implications. It would be possible to eliminate the need for a left join by dividing the hash space so there was no overlap between the hash values of resources and of literals, and then storing them both in the same table. In practice, however, the left joins do not appear to incur that great a performance penalty.

While dividing the hash space would remove the requirement for a left join it would require storing the resources and literals in the same SQL table, which has both design cleanliness and performance implications of its own. As literals and resources are disjoint first class objects in the model theory, inserting them in a single table causes confusion at the database schema level. In addition, it would be inefficient if we were to store further data about resources or literals, such as the bNode/uriref condition of a resource or the datatype of a literal, because this would require the existence of columns that would be of use for only one type and unnecessarily increase the size of the rows.

RDQL constraints, such as that shown in the second RDQL example, can be expressed directly as relational constraints. The second RDQL example is transformed as follows:

$$
\begin{aligned}
\{r.uri \mid \text{resource}(r) \ \wedge \ (\exists t_1)(\ & \text{triple}(t_1) \wedge \\
& t_1.subject = r.hash \wedge \\
& t_1.predicate = \text{hash(p:a)} \wedge \\
& \text{not}(t_1.object = \text{hash(foo)})\ )\}
\end{aligned}
$$

In this relational expression, the `?c != "foo"` constraint from the RDQL expression simply becomes not($t_1.object$ = hash(foo)). Finally, the MySQL optimiser efficiently handles any trivial constraints (such as that shown below), without needing to rewrite the query into an equivalent RDQL expression in which all occurrences of `?c` have been replaced by `"foo"`.

```
SELECT ?a
WHERE  (?a, <p:a>, ?c)
       (?a, <p:b>, ?c)
AND    ?c == "foo"
```

4.6 Datatypes

The introduction of RDF datatypes makes the query transformation process more complex, because we require that a datatype be explicitly supported by the database in order to handle it efficiently within the RDBMS engine. For simple datatypes such as integers and floating point numbers this should not cause problems, but more complex types such as dates may prove difficult, as mentioned in Section 4.5.

Basic datatype handling could be added by using the RDBMS' runtime conversion features, storing integers, for example, in an textual representation (UTF8 in the case of MySQL) and using the SQL integer comparison operators `<`, `>` and `=` with integer constants to perform the RDQL equivalents. This would most likely be inefficient because it does not allow the database's native indexing to accelerate the comparison; the index is over the textual representation, not the native integer representation.

Alternatively, the literals table could include a union representation of all the types supported, either flagged by some discriminator variable, or by storing each type in a separate table and joining them as required. The former would lead to a large row size for the literal table and inefficient storage for numeric types which require little space, and the latter would lead to complex join expressions when the type of the literal was unknown at query time. The evaluation of these approaches to discover the best trade-off is a suitable area for future work.

4.7 Hybrid eager/lazy production

As described in Section 2.3, the model theory for RDF and RDFS [16] contains a number of entailments which should be generated by an RDF inference engine. Existing systems generate these in several ways, but the most common approach employs a production rule system which generates the required entailments by either forward chaining from asserted facts or backward chaining from queries that are presented to the system.

These approaches both have advantages and disadvantages: a purely forward chaining system applies the entailment rules in the model theory exhaustively to the asserted facts in order to generate the RDF closure (the deductive closure) of those facts. This eager evaluation of the deductive closure (performed by systems such as Sesame) has the effect of reducing the processing cost of evaluating queries; the entailments which will be returned by a query (in addition to any ground facts) are generated ahead of time, which may reduce the time taken to evaluate queries in some cases (a considerable advantage for interactive applications where a fast response time is of paramount importance). The disadvantage of this approach is that the RDF closure may be many times the size of the asserted facts, and contain many entailments which match queries only rarely.

Conversely, a purely backward chaining system evaluates the entailments matched by a query at query processing time. This reduces the cost of storing the entailments (since only those that are needed are generated) at the cost of more expensive query processing; a backward chaining system that evaluates entailments lazily might incur a time penalty that makes its use in interactive applications unworkable. Although the cost of repeated queries may be reduced by caching the generated entailments, there still exists a substantial cost at query time.

We adopt a hybrid approach in which some entailment rules – typically those which generate fewer entailments – are evaluated eagerly using forward chaining rules when new facts are asserted, while those with a greater storage cost and a lower evaluation cost are evaluated as required at query time by a combination of backward chaining and query rewriting. A summary of which entailments are generated eagerly and which are generated lazily is given in the column labelled 'Eval' in Table 1.

By producing some of the entailments lazily, it is possible to trade triple table size and assertion time complexity for query time complexity. In general we have found this to be advantageous, especially in the case of rdfs6 (demonstrated below), which can significantly increase the cost of A-box assertions if entailment production is done eagerly. In cases where the T-box is large and the A-box small, it is likely to be more efficient to produce entailments eagerly.

For example, rdfs6 (the application of the transitive closure of the rdfs:subPropertyOf relation to all other triples) is as follows:

$$\begin{array}{l} \text{xxx aaa yyy} \\ \text{aaa rdfs:subPropertyOf bbb} \end{array} \Rightarrow \text{xxx bbb yyy}$$

This entailment may be generated at query time by back-chaining the xxx bbb yyy expression, such that `(a, b, c)` is re-written as `(a, ?p, c), (?p, rdfs:subPropertyOf, b)` in RDQL syntax. Furthermore, if bound, the predicate `b` may be examined at query time to establish whether it has any sub-properties, avoiding the rewrite and consequent additional relational join where it would be unnecessary.

We can apply this approach in a similar manner to rdfs9, which applies the transitive closure of the rdfs:subClassOf relation to triples containing the rdf:type property.

$$\begin{array}{l} \text{xxx rdfs:subClassOf yyy} \\ \text{aaa rdf:type xxx} \end{array} \Rightarrow \text{aaa rdf:type yyy}$$

Queries affected by this entailment rule may be rewritten, such that `(a, <rdf:type>, c)` becomes `(?t, rdfs:subClassOf, c), (a, rdf:type, ?t)`.

Detecting when this query rewriting is necessary becomes more complex when the predicate of a given triple pattern is a free variable, such as `SELECT ?p WHERE (?s, ?p, ?o)`, as the free variable `?p` may match one of the entailment rules. In this case, the query component for this triple expands to the following disjunctive expression:

$$\begin{array}{rl} \{r.uri \mid resource(r) \wedge & \\ ((\exists t_1)(& \text{triple}(t_1) \wedge \\ & t_1.predicate = r.hash\ ) \vee \\ (\exists t_2)(\exists t_3)(& \text{triple}(t_2) \ \wedge\ \text{triple}(t_3) \wedge \\ & t_2.subject = t_3.object \wedge \\ & t_2.predicate = \text{hash}(\text{rdfs:subClassOf}) \wedge \\ & t_3.predicate = \text{hash}(\text{rdf:type}) \wedge \\ & t_3.predicate = r.hash\ ))\} \end{array}$$

If only the first (most obvious) clause of this disjunctive expression is used as the expansion of the query, potential entailments for rdfs9 will be missed.

The choice of which entailments may be generated lazily is heavily dependent on the relationship between the consequent of one entailment rule and the antecedent of another. We cannot lazily evaluate a rule which generates entailments that may trigger other rules that are evaluated eagerly and still perform complete reasoning. Consequently, we have chosen to lazily evaluate both rdfs6 and rdfs9, which respectively depend on the (eagerly evaluated) transitive closures of rdfs:subPropertyOf and rdfs:subClassOf.

4.8 Transitive entailments

Certain RDFS entailment rules (such as rdfs5a) express the transitive nature of some properties in the RDFS vocabulary. We cannot lazily construct the transitive closures of these properties using a simple SQL join in MySQL, so currently we can only generate the entailments for them eagerly. We have implemented these transitive entailments using an optimised implementation of the TRANSITIVE-CLOSURE function [11].

The SQL–99 language [13] defines a syntax for expressing transitive joins, with which it would be possible to compute transitive entailments lazily. However, the existing SQL–99 implementations for transitive joins are not very efficient, and so are unsuitable for lazy evaluation if interactive performance is required [25].

5 Assertion performance

We have not expended any significant effort in optimising the assertion of RDF in 3store's because we have found the existing performance to be adequate. On commodity hardware, 3store can assert between 1000 and 300 triples/second depending on the size of the knowledge base. We have performed some simple optimisations, such as optionally locking the tables to prevent index rebuilding and grouping SQL INSERT statements, but further work in this area will be undertaken.

6 Availability

The 3store software is available under the GNU General Public Licence from Sourceforge at `http://sourceforge.net/projects/threestore`.

7 Conclusions

In this paper, we have described our RDF engine, 3store, which efficiently supports the RDF and RDFS entailments over relatively large RDF knowledge bases using a relational database back-end to perform the queries. This makes certain assumptions about the RDF data being stored: the T-box should be significantly smaller than the A-Box, and the contents of the knowledge base relatively stable such that large portions of the data will not change with high frequency.

The development of 3store is an ongoing endeavour with considerable scope for further work. We intend to extend our study of the indexing schemes for the tables used to represent triples in this schema, because the choice of indexes can dramatically affect the query or assertion time. We have already refined our relational table layout from that used in the prototype, but more work is necessary here because it could be that there are untried schemata that would yield better performance. Our future plans for 3store include improving the performance of RDF assertions, some limited support for OWL features (chiefly `owl:sameAs`, which is needed to describe coreferent resources when combining different RDF sources [4]).

Our experiences with implementing 3store and evaluating existing systems show that there is a lack of standardised performance benchmarks for RDF knowledge bases. The W3C has already produced some test cases for RDF entailments [15] and RDF query languages [8], but these could be extended to include query and assertion performance which would be useful in evaluating the strengths and weaknesses of the various RDF knowledge bases.

8 Acknowledgements

This work is supported under the Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (IRC), which is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01. The AKT IRC comprises the Universities of Aberdeen, Edinburgh, Sheffield, Southampton and the Open University.

References

1. N.F. Abernethy and R.B. Altman, *Sophia: Providing basic knowledge services with a common DBMS*, Proceedings of the 5th International Workshop on Knowledge Represenation Meets Databases (KRDB '98): Innovative Application Programming and Query Interfaces (1998).
2. *Advanced Knowledge Technologies*, `http://www.aktors.org/`, 2000.
3. *The AKT Reference Ontology*, `http://www.aktors.org/publications/ontology/`, 2002.
4. Harith Alani, Srinandan Dasmahapatra, Nicholas Gibbins, Hugh Glaser, Steve Harris, Yannis Kalfoglou, Kieron O'Hara, and Nigel Shadbolt, *Managing reference: Ensuring referential integrity of ontologies for the semantic web*, Proceedings 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW'02), 2002, pp. 317–334.
5. Dave Beckett, *Scalability and Storage: Survey of Free Software/Open Source RDF storage systems*, ILRT Technical Report **1016** (2002), `http://www.ilrt.bris.ac.uk/publications/researchreport/rr1016/`.
6. ______, *Raptor RDF Parser Toolkit*, `http://www.redland.opensource.ac.uk/raptor/`, 2003.
7. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway, *UMAC: Fast and secure message authentication*, Advances in Cryptology - CRYPTO '99. Lecture Notes in Computer Science **1666** (1999), 216–233, `http://www.cs.ucdavis.edu/~rogaway/papers/umac-proc.ps`.
8. Dan Brickley, *RDF Query and Rule Testcase Repository*, `http://www.w3.org/2003/03/rdfqr-tests/`, 2003.
9. Jeen Broekstra and Arjohn Kampman, *Sesame: A generic architecture for storing and querying RDF and RDF Schema*, Technical report, Aidministrator Nederland b.v., October 2001, `http://sesame.aidministrator.nl/publications/del10.pdf`.
10. Vinay Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice, *Open knowledge base connectivity*, Tech. report, OKBC Working Group, April 1998, `http://www.ai.sri.com/~okbc/spec.html`.
11. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algortihms*, MIT Press, Cambridge, MA, 1989.
12. Scott A. Crosby and Dan S. Wallach, *Denial of service via algorithmic complexity attacks*, USENIX Security (2003).
13. International Organisation for Standardisation: Information Technology Database Languages SQL, *Part 2: Foundation*, ISO/IEC 9075-2:1999, International Organisation for Standardisation, 1999.
14. Free Software Foundation, *GNU General Public License*, `http://www.gnu.org/licenses/gpl.txt`, 1991.
15. Jan Grant and Dave Beckett, *RDF test cases*, Working draft, World Wide Web Consortium, January 2003, `http://www.w3.org/TR/rdf-testcases/`.

16. Patrick Hayes, *RDF Semantics*, Working draft, World Wide Web Consortium, January 2003, `http://www.w3.org/TR/rdf-mt/`.

17. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl, *RQL: A declarative query language for RDF*, Proceedings of the 11th International World Wide Web Conference (WWW2002), 2002.

18. Hewlett-Packard Labs, *RDQL - RDF data query language*, `http://www.hpl.hp.com/semweb/rdql.htm`, 2003.

19. ______, *The Jena Semantic Web Toolkit*, Web page, Hewlett-Packard Labs, 2003, `http://www.hpl.hp.com/semweb/jena.htm`.

20. Maryland Information and Network Dynamics Lab, *ParkaSW*, `http://www.mindswap.org/2002/parka/`, 2003.

21. Deborah L. McGuinness and Frank van Harmelen, *OWL Web Ontology Language Overview*, Working draft, World Wide Web Consortium, March 2003, `http://www.w3.org/TR/owl-features/`.

22. Libby Miller, Andy Seaborne, and Alberto Reggiori, *Three implementations of SquishQL, a simple RDF query language*, Tech report, Hewlett-Packard Labs, 2002, `http://www.hpl.hp.com/techreports/2002/HPL-2002-110.html`.

23. M.C. Norrie, U. Reimer, P. Lippuner, M. Rhys, and H.J. Schek, *Frames, objects and relations: Three semantic levels for knowledge base systems*, Reasoning About Structured Objects: Knowledge Representation Meets Databases. In 1st Workshop KRDB'94 (1994), 20–22.

24. R. Rivest, *The MD5 message-digest algorithm*, IETF RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992, `http://www.ietf.org/rfc/rfc1321.txt`.

25. Guther Schadow, Michael Barnes, and Clement McDonald, *Representing and querying conceptual graphs with relational database management systems is possible*, Processinds of AMIA Syposium 2001:598-602 (2001).

26. Nigel R. Shadbolt, monica m.c. schraefel, Nicholas Gibbins, and Stephen Harris, *CS AKTive Space: or how we stopped worrying and learned to love the semantic web*, `http://eprints.ecs.soton.ac.uk/archive/00007440/`, 2003.

27. Kilian Stoffel, Merwyn Taylor, and James Hendler, *Efficient management of very large ontologies*, Proceedings of American Association for Artificial Intelligence Conference (AAAI-97) (1997).

28. TAP Project, *GetData: Querying the Data Web*, `http://tap.stanford.edu/tap/getdata.html`, 2003.